Customizing Manual

ADITO Software GmbH

Version 2024.1.5 | 2025-03-20

Version	Changes
2024.1.5	• New chapter Consumer filter • Chapter Context filter (content search): Property "isLookupFilter" mentioned. • Chapter Automatisms: New sub-chapter Visibility of drawers. • Various minor optimizations.
2024.1.4	• Chapter Debugging vs. temporary logging improved and enhanced. • Chapter Liquibase update: Updated screenshot and text, including an additional remark, regarding option "example". • Bugfix: Names of various libs updated. • References to AID123 Modularization included. • Various minor optimizations.

Summaries of changes in previous versions of this document can be found in appendix Version history.

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

Preface

This document enables you to build and customize ADITO applications for multiple purposes. The first part of this manual is designed like a schoolbook: On the basis of a plain example, you learn to handle ADITO step-by-step. It is not recommended to skip one of these chapters, as each chapter implies that you have read the previous ones.

The second part of this manual is more glossary-like: Additional knowledge is imparted using various best-practice examples included in the ADITO xRM project. Further helpful details are available in the appendices.

Happy reading!

The ADITO Academy

1. Introduction

ADITO is a comprehensive software framework that enables you to build powerful web-based xRM solutions, with **x**RM standing for "any **r**elationship", not only the relationships to customers (CRM). An ADITO distribution includes the "ADITO Designer", which is ADITO’s IDE for customizing (creating new applications and modifying existing ones). Furthermore, ADITO comes with a project called "ADITO xRM", which includes the most important data structures to manage companies, contact persons, activities, sales-related elements (offers etc.), marketing campaigns, and many more. Besides, the ADITO xRM project also includes some example data, e.g., companies and contact persons.

The central programming paradigm of ADITO is the so-called "Neon Programming Model", which is based on the "Entity model". This concept means a strict separation of how data is

- structured and calculated ("Entities")
- stored ("RecordContainer") and
- displayed ("Views", clustered in "Contexts").

The **benefits** of this approach are:

- The same data or logic can be displayed in various ways, using different Views.
- Views can be (re-)used in multiple Contexts.
- The obligatory usage of ViewTemplates ensures a uniform, consistent appearance of the ADITO application. Still, users have some options to customize the presentation of data according to their requirements, e.g., by arranging various Views on a Dashboard.
- If required, the data source can easily be replaced later.

- The web-based concept simplifies the usage: Once the ADITO server has been installed, the users only need a chromium-based browser and a hyperlink to the server in order to run ADITO. This also ensures an easy software maintenance: Updates of ADITO only need to be performed on the server, not on the users' workstations.
- Development can easily be divided into "frontend" and "backend" programming.

In this manual, you will learn how to build ADITO applications on your own. The teaching approach is process-oriented, i.e., after an overview you will be introduced to handling the ADITO Designer step-by-step, following the usual development process.

2. Overview

2.1. Structure of ADITO projects

At first, you will **structure your data into business objects**, which have different features. This structure will then be formed into **Entities**, which have different features, the so-called **EntityFields** - e.g. the Entity "Person_entity" features the EntityFields "FIRSTNAME", "LASTNAME", and "DATEOFBIRTH".

The next step is to **set up the data source**, which usually is a **database**: Every Entity gets its data from a specific data source. In most cases, this data source is a database table, with most of the EntityFields each corresponding to one specific column of this table - except for fields that are calculated through other fields or have other sources. The database tables and fields can either be created manually (via SQL scripts, database managment systems, or via ADITO’s built-in database editor), or you can use a third-party tool called **Liquibase**, in order to create database elements (tables, columns, etc.) as well as database content, using xml configuration files.

After the database has been set up, its structure of tables and fields (but not their data content itself) needs to be referenced in the ADITO project. This can be considered as a kind of "copy" of the database structure and is therefore called "the database structure of the project" or "**Alias Definition**". This construct enables you, if required, to augment the references to database tables and columns with additional features, e.g., a description, a documentation, or specific properties.

The next step is to connect every field of the Data alias with its respective EntityField. This mapping is configured in an ADITO model called "**RecordContainer**", which you can treat as a kind of "interface" (or "adapter", to be more precise) between the ADITO project and its data source.

As Entities can have relations to each other (e.g., the "Organisation_entity" is related to the "Person_entity", which represents persons, who may work in an organisation), we need to connect them in order to make data exchange possible. In ADITO, these mechanisms are realized by configuring so-called "**Providers**" (offering datasets to other Entities, partly based on specific "Parameters") and "**Consumers**" (using the passed "Parameters" and processing the datasets offered via a Provider).

These steps all belong to the non-visible part of the ADITO application ("backend"). They are prerequisites for building the visible part ("frontend"). This visible part mainly consists of so-called "**Views**", which are visual components used to display data of one or more Entities in a structured way (with "View" spelled with a capital letter, in order to distinguish it from "view" in general language use.). The appearance of every View is determined by one or more **ViewTemplates**. Users do not need to build ViewTemplates by themselves, but they can select suitable templates from a pool of several **ViewTemplateTypes** predefined by ADITO (e.g., a table, a list, or a Gantt chart, and many more). This restriction to a limited number of ViewTemplateTypes ensures a user-friendly, intuitive, and thus easy-to-learn handling of ADITO applications as well as a uniform, consistent look-and-feel. Each ViewTemplate references specific **EntityFields**.

Every View belongs to one specific Entity. One Entity can have zero to multiple Views. All Views of an Entity are clustered in a so-called "**Context**". (Spelled with a capital letter in order to distinguish it from "context" in general language use.) A Context usually contains at least the following standard Views:

- The **FilterView** is the entry point of a Context. It usually shows all data in a table or as a tree. To the right of it, it has a filter component allowing to restrict the data according to specific filter criteria.
- The **PreviewView** is shown to the right of the FilterView, when you click on one of its datasets. The PreviewView shows specific detail features of the dataset selected in the FilterView.
- The **MainView** is reached by marking one dataset of the FilterView and clicking on the "open" button. It shows the PreviewView as "master dataset" on the left, and a tabbed component on the right ("detail" part), which represents referenced Views of other Contexts (e.g., a company is displayed in the PreviewView, and the persons working in this company are displayed on the right).
- The **EditView** is opened via the "Plus sign" button or "Pencil" button, respectively, allowing to create new datasets or to modify existing datasets.

Furthermore, a View can be extended by adding **references to other Views** (of the same Context or of other Contexts). Thus, Views are re-usable. For example, a View of the Context "Organisation" may include references to Views of the Contexts "Person" (showing the employees of the organisation), or "Activity" (showing activities referring to the organisation, e.g., a phone call or a visit).

Finally, the Contexts and their Views must be made selectable in a specific menu group of the Global Menu of the ADITO frontend (web client). This can easily be done in a **menu editor** window of the ADITO Designer, simply by dragging and dropping a Context from a pool of Contexts to a specific place of the Global Menu.

Additionaly, you can define **user roles** and assign them to both the users and the different menu items. This restricts the visible menu items to ADITO users with specific user roles.

Despite the usage of ViewTemplates, users can partly modify the ADITO application according to their requirements. In particular, you can create an overview of the data most important to you by arranging multiple Views from different Contexts on a **Dashboard**. Furthermore, e.g., in Views containing tables, you can change the column width and the sorting of the datasets.

Last but not least, ADITO is perfectly suitable for being applied in an **international** context. Every textual element in its components can automatically be translated into the user’s language.

From version 2024.2, the ADITO xRM project is structured in a **modularized** way. Working with modules increases flexibility and reduces merging efforts for updates of xRM in customer projects. However, to simplify matters, the example task in this manual is not realized as a module. If you are interested in details about modularization in ADITO, you may read the ADITO Information Document AID123 Modularization.

2.2. Logical hierarchy

To sum up, ADITO projects are structured by the following logical hierarchy:

- Each project has *one* **Global Menu**.
- The Global Menu consists of one or multiple **menu groups**.
- Each menu group consists of one or multiple **Contexts**.
- Each Context has
  - ***one* Entity** assigned, which has one or multiple EntityFields.
  - one or multiple **Views**.
- Each View consists of one or multiple **ViewTemplates** (or of references to other Views), with each of it based on one specific **ViewTemplateType**.
- Each ViewTemplate references one or multiple **EntityFields**.

This logical hierarchy determines both the visual appearance of the web client and the structure of the project source data, as visible in the ADITO Designer:

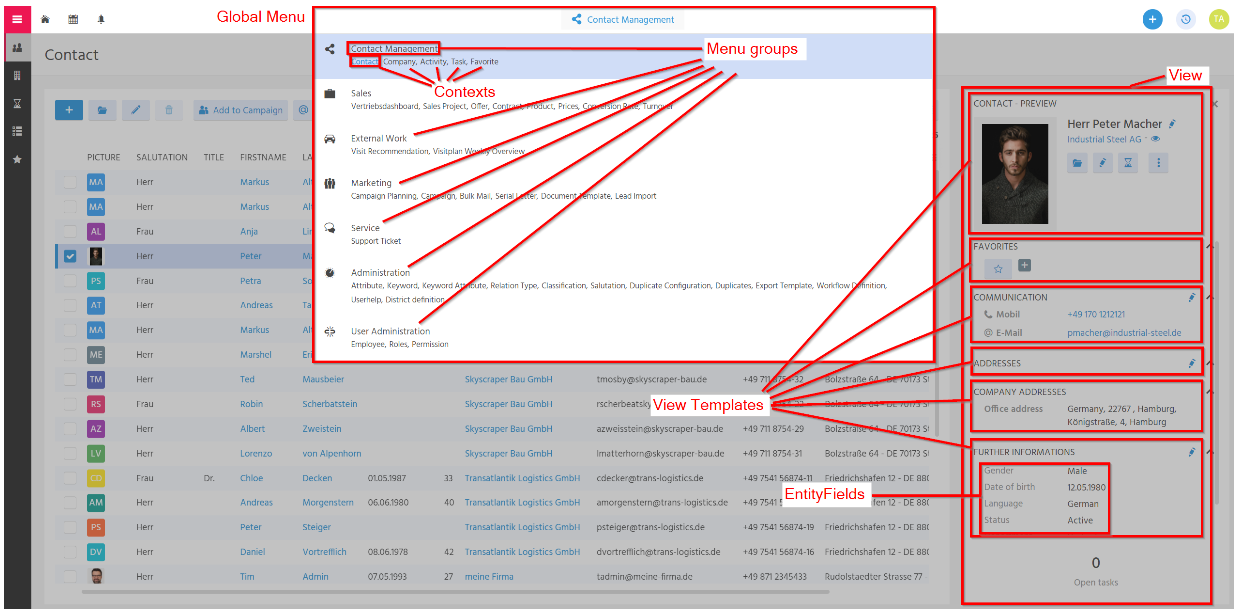

Global Menu
Menu groups
Contexts
View
View Templates
EntityFields
Contact Management
Contact
Company, Activity, Task, Favorite
Sales
Vertriebsdashboard, Sales Project, Offer, Contract, Product, Prices, Conversion Rate, Turnover
External Work
Visit Recommendation, Visitplan Weekly Overview
Marketing
Campaign Planning, Campaign, Bulk Mail, Serial Letter, Document Template, Lead Import
Service
Support Ticket
Administration
Attribute, Keyword, Keyword Attribute, Relation Type, Classification, Salutation, Duplicate Configuration, Duplicates, Export Template, Workflow Definition, Userhelp, District definition
User Administration
Employee, Roles, Permission
CONTACT - PREVIEW
Herr Peter Macher
Industrial Steel AG
FAVORITES
COMMUNICATION
Mobil +49 170 1212121
E-Mail pmacher@industrial-steel.de
ADDRESSES
COMPANY ADDRESSES
Office address Germany, 22767, Hamburg, Königstraße, 4, Hamburg
FURTHER INFORMATIONS
Gender Male
Date of birth 12.05.1980
Language German
Status Active
0
Open tasks

Figure 1. Hierarchy of elements in the ADITO web client

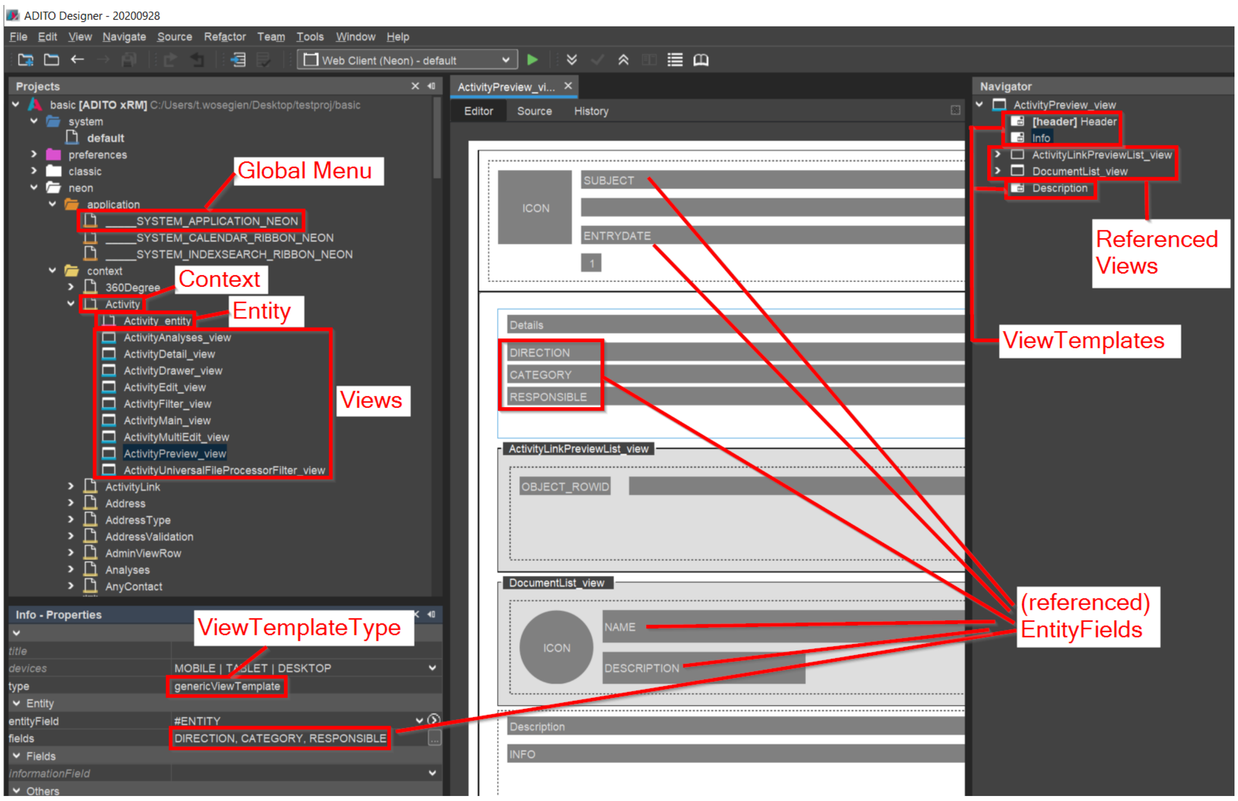


Figure 2. Structure of elements in the ADITO Designer

In the following chapters, we will go through the above outlined development steps in detail. For a better visualization and for practice, each step is explained using the **example of a company car pool** with several cars, several drivers, several reservations, and some more options.

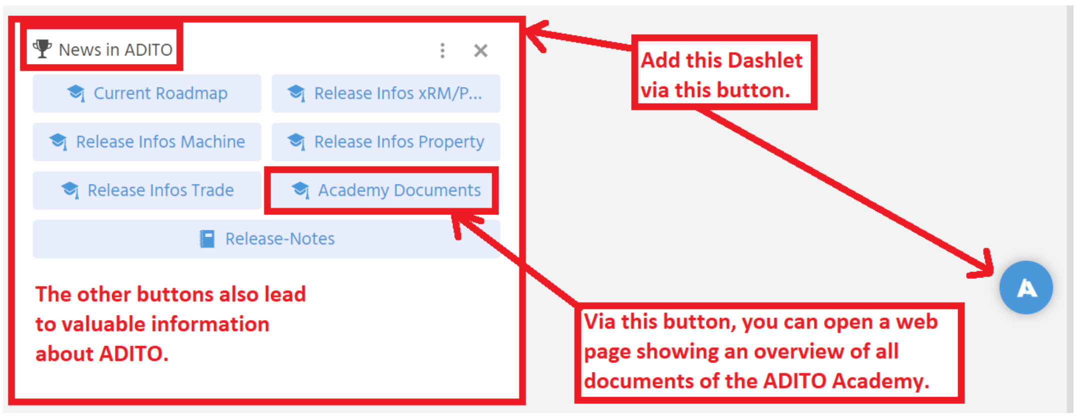
3. Prerequisites

This manual is designed as a schoolbook, requiring active participation of the reader. After the introductory chapters, you should reproduce the examples given in the following chapters with an ADITO system running on your own computer. Thus, the prerequisites for reading on are as follows.

3.1. Documentation

Most of the ADITO documentation is available

- for download from the customer area of the ADITO website.
- via the overview page "Academy Documents", available via the ADITO Service Client. You can open this page
  - directly via this link;
  - by adding the Dashlet "News in ADITO" and there clicking on button "Academy Documents":


News in ADITO
Current Roadmap
Release Infos xRM/P...
Release Infos Machine
Release Infos Property
Release Infos Trade
Academy Documents
Release-Notes
The other buttons also lead to valuable information about ADITO.
Add this Dashlet via this button.
Via this button, you can open a web page showing an overview of all documents of the ADITO Academy.

For a good understanding of the Customizing Manual, you should be familiar with the following documents (or at least have them available on demand):

- Designer Manual
- ADITO Information Documents (AID), in particular the following:
  - Coding Styles | AID001-EN
  - Wording Guideline | AID002-DE
  - Design Guideline | AID003-EN
  - Performance Optimization | AID066-EN

Some ADITO documents, like this Customizing Manual, contain **code snippets**, which you can copy into your own ADITO project or use to verify the correctness of your own code. We recommend

- to use the latest version of Adobe Acrobat Reader DC in order to view this manual, because the usage of other PDF readers can result in problems when copying code from PDF file into your project (additional special characters or formatting characters may be inserted then, which results in a failure of the code);
- to apply the automated code formatting (shortcut: SHIFT+ALT+F) after copying and inserting the code;
- **to remove additional line breaks, which might have been inserted by this manual's PDF generator (especially at long lines), as these can make the code invalid.**

3.2. ADITO Web Client

To ensure an efficient customizing work, you should be familiar with the ADITO Web Client and know at least the basic functionality of the ADITO xRM project from a client user's perspective. You can learn this

- in the "web client" part of the ADITO training course for developers (with even deeper client-related training courses being available on request);
- by reading the presentations for client users, available via paragraph "Anwendungspräsentationen" (user presentations) in the overview page "Academy Documents", available via the ADITO Service Client;
- by simply browsing through the various menu groups and Contexts of the ADITO xRM project included in the ADITO test system that ADITO has provided you with.

> You need to use a Chromium-based browser, such as Google Chrome or Microsoft Edge. Other types of browsers are not supported.

3.3. ADITO platform and xRM project

This manual assumes that you are working with an ADITO cloud system, including the project "ADITO xRM", and with a local ADITO Designer (this is the name of ADITO's IDE for customizing). Usually, ADITO will provide you with

- an ADITO cloud system, including the ADITO xRM project
- all relevant access credentials
- a compressed file (zip), including the ADITO Designer, which you simply have to unpack at an arbitrary place of your local hard disk/SSD
- an installation guide about how to prepare and start your local ADITO Designer and then open ("load") your ADITO xRM project in the Designer.

Once you have completed this setup, you can start with your productive customizing work, or with configuring the "car pool" example in this manual, respectively.

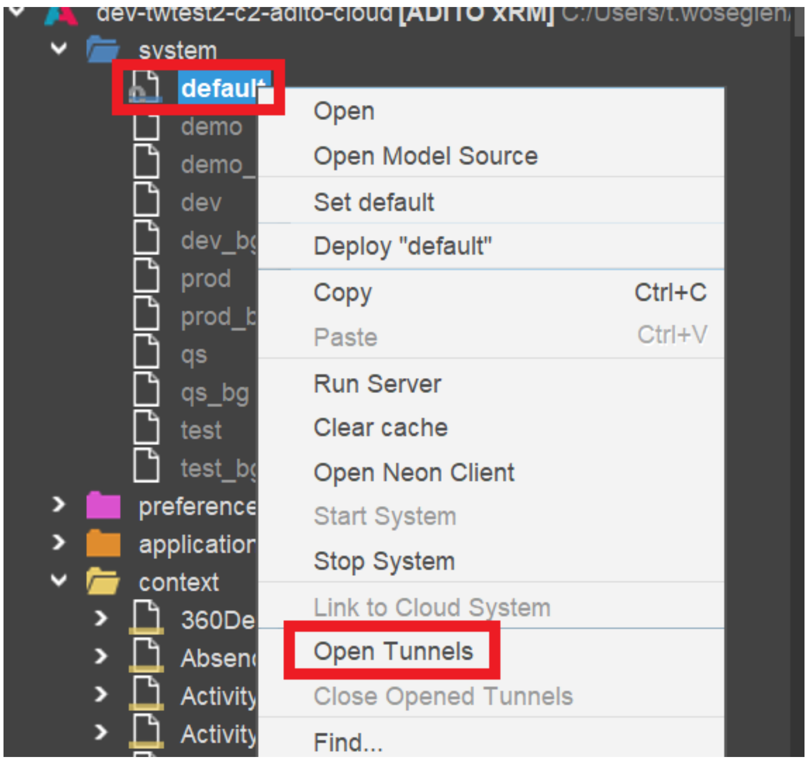
Being familiar with the basic functionality of the ADITO Designer is prerequisite for reading this manual. Find more information in the Designer Manual.

If required, ask your ADITO contact for further instructions.

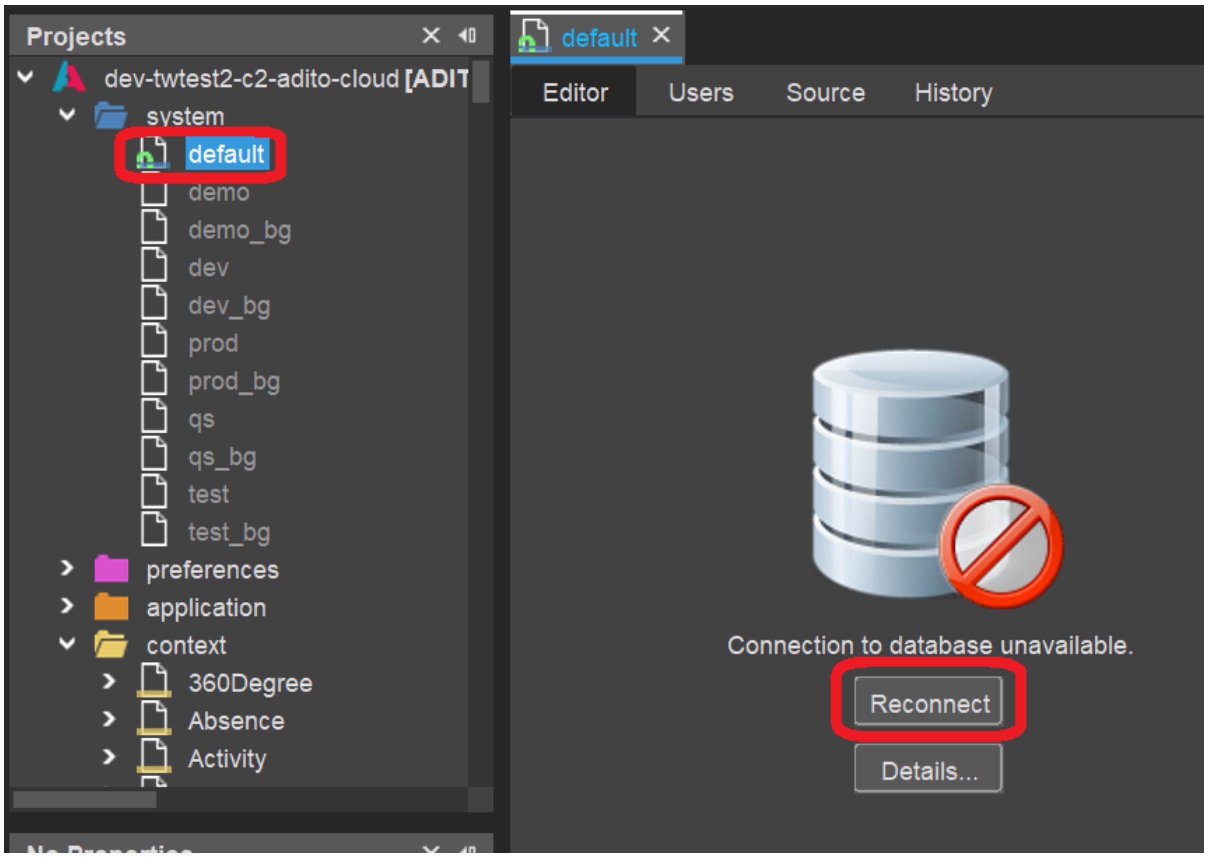
3.4. ADITO database

This manual uses "MariaDB" for teaching purposes, as this is the default database in ADITO cloud systems.

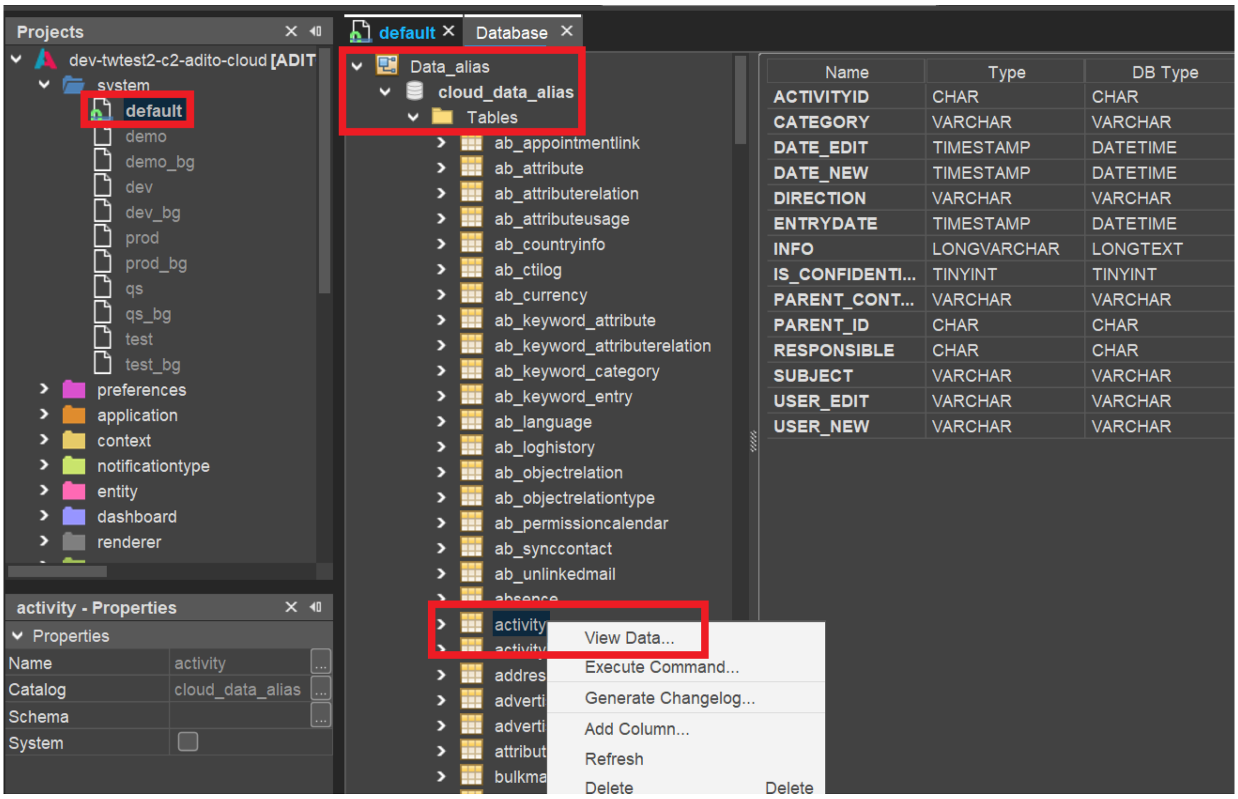
You can connect your ADITO Designer with the database of your cloud system via tunneling: In the "Projects" window, right-click on "system" > "default" and choose "Open tunnels" from the context menu.



After a few seconds, the tunnel icon to the left of "default" becomes green. If you then double-click on "default" (under "system"), the Editor window will show a tab named "default", including, amongst others, an entry for the system database ("____SYSTEMALIAS") and for the data database ("Data_alias"). (Sometimes, you need to click button "Reconnect" in tab "default" first).



You may double-click on the system database or the data database and open schema "ADITO", in order to inspect the ADITO-related tables included in these databases. If you right-click on a table, you may, e.g., choose "View Data...",



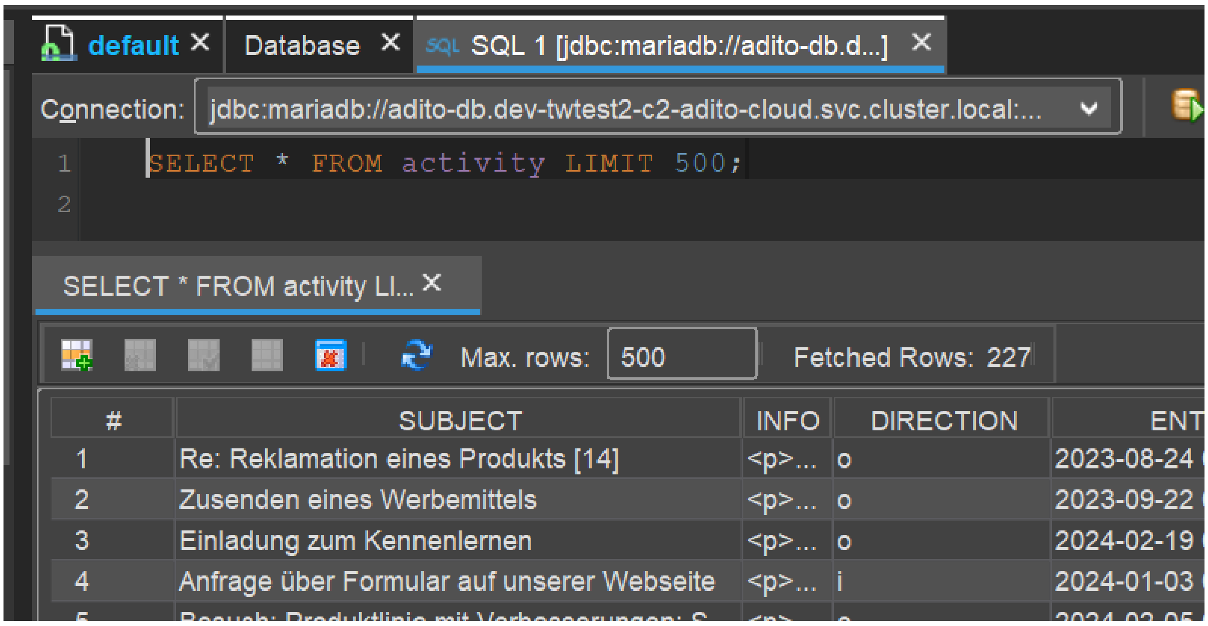
Name	Type	DB Type
ACTIVITYID	CHAR	CHAR
CATEGORY	VARCHAR	VARCHAR
DATE_EDIT	TIMESTAMP	DATETIME
DATE_NEW	TIMESTAMP	DATETIME
DIRECTION	VARCHAR	VARCHAR
ENTRYDATE	TIMESTAMP	DATETIME
INFO	LONGVARCHAR	LONGTEXT
IS_CONFIDENTI...	TINYINT	TINYINT
PARENT_CONT...	VARCHAR	VARCHAR
PARENT_ID	CHAR	CHAR
RESPONSIBLE	CHAR	CHAR
SUBJECT	VARCHAR	VARCHAR
USER_EDIT	VARCHAR	VARCHAR
USER_NEW	VARCHAR	VARCHAR

activity - Properties

Properties

Name	activity
Catalog	cloud_data_alias
Schema	
System	☐

then an SQL "select" statement will automatically be generated and executed, showing you the datasets of this table.


default
Database
SQL 1 [jdbc:mariadb://adito-db.d...]
SELECT * FROM activity LI...

Connection: jdbc:mariadb://adito-db.dev-twtest2-c2-adito-cloud.svc.cluster.local:...

```
SELECT * FROM activity LIMIT 500;
```

Max. rows: 500 Fetched Rows: 227

#	SUBJECT	INFO	DIRECTION	ENT
1	Re: Reklamation eines Produkts [14]	<p>...	o	2023-08-24
2	Zusenden eines Werbemittels	<p>...	o	2023-09-22
3	Einladung zum Kennenlernen	<p>...	o	2024-02-19
4	Anfrage über Formular auf unserer Webseite	<p>...	i	2024-01-03
5	Besuch: Produktlinie mit Verbesserungen: S	<p>...	o	2024-02-05

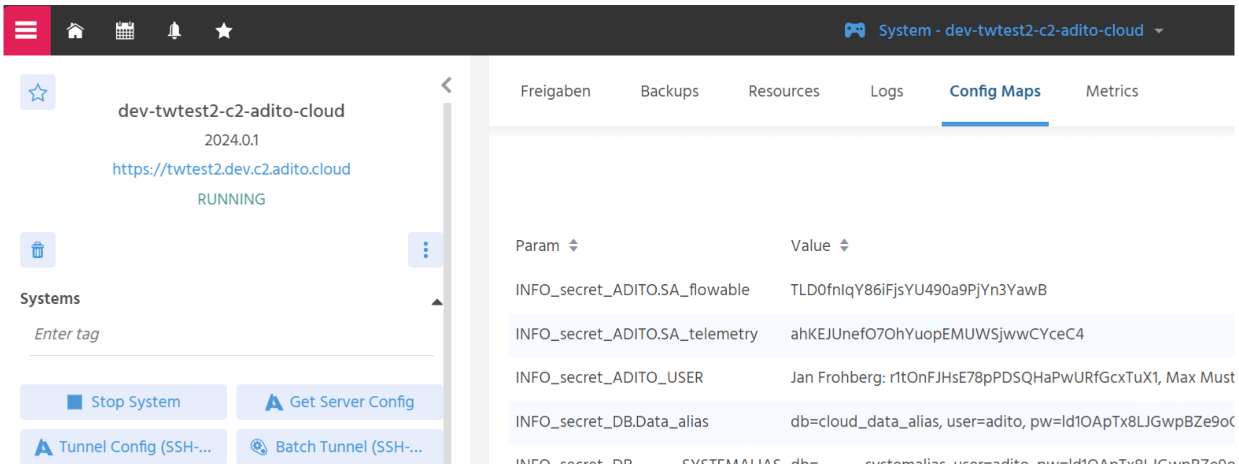
Find more information in the ADITO Designer Manual. If required, ask your ADITO contact for further instructions.

3.5. ADITO server

The ADITO server manages the communication between the ADITO databases ("system database" and "data database") and the client browser: It reads

- the data defining the visual elements (e.g., a table structure, or a chart's layout) from the "system database" - in particular, from table ASYS_SYSTEM;
- the productive data (i.e., the datasets to be shown in a table) from the "data database", e.g., from table ACTIVITY (see screenshot above)

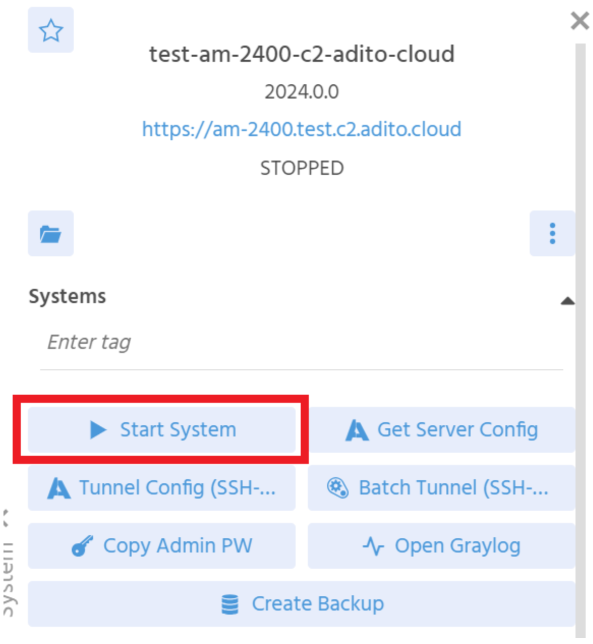
In ADITO's Self-Service Provider (SSP, see installation guide), you can see, if the server of your ADITO system is running (status "RUNNING") or shut down (status "STOPPED"). Furthermore, in the MainView of the SSP's Context "System", you can see various details about an ADITO system's server:



Param	Value
INFO_secret_ADITO.SA_flowable	TLD0fnIqY86iFjsYU490a9PjYn3YawB
INFO_secret_ADITO.SA_telemetry	ahKEJUnefO7OhYuopEMUWSjwwCYceC4
INFO_secret_ADITO_USER	Jan Frohberg: r1tOnFJHsE78pPDSQHaPwURfGcxTuX1, Max Must
INFO_secret_DB.Data_alias	db=cloud_data_alias, user=adito, pw=ld1OApTx8LJGwpBZe9o

Figure 3. The SSP's MainView of an ADITO system

If the server of your ADITO system is not running, start it now, via button "Start System":


test-am-2400-c2-adito-cloud

2024.0.0

https://am-2400.test.c2.adito.cloud

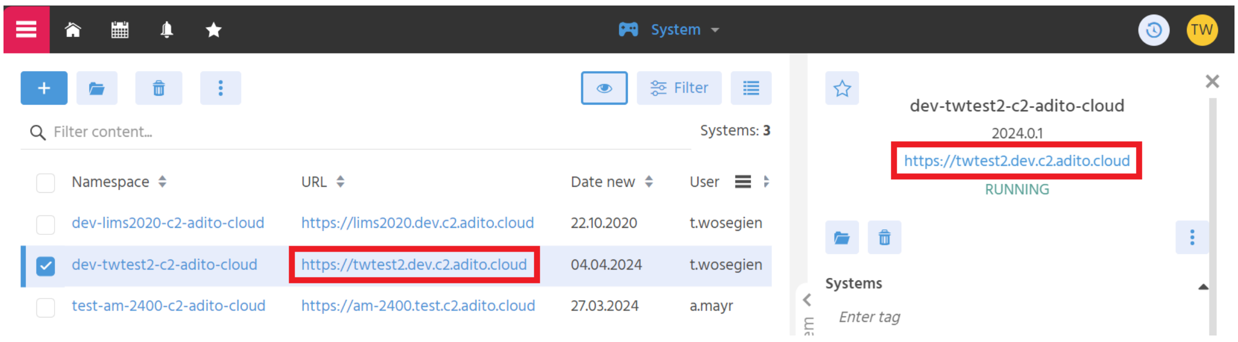
STOPPED

Systems

Enter tag

Wait a few minutes, until the server’s startup is completed. (Press the "Refresh" button of your browser, in order to update the display of the status.)

Once the server has status "RUNNING", you can login to your system’s ADITO xRM project, as shown in the web client, by using the URL displayed in the SSP:



Systems: 3

Namespace	URL	Date new	User
dev-lims2020-c2-adito-cloud	https://lims2020.dev.c2.adito.cloud	22.10.2020	t.wosegien
dev-twtest2-c2-adito-cloud	https://twtest2.dev.c2.adito.cloud	04.04.2024	t.wosegien
test-am-2400-c2-adito-cloud	https://am-2400.test.c2.adito.cloud	27.03.2024	a.mayr

dev-twtest2-c2-adito-cloud

2024.0.1

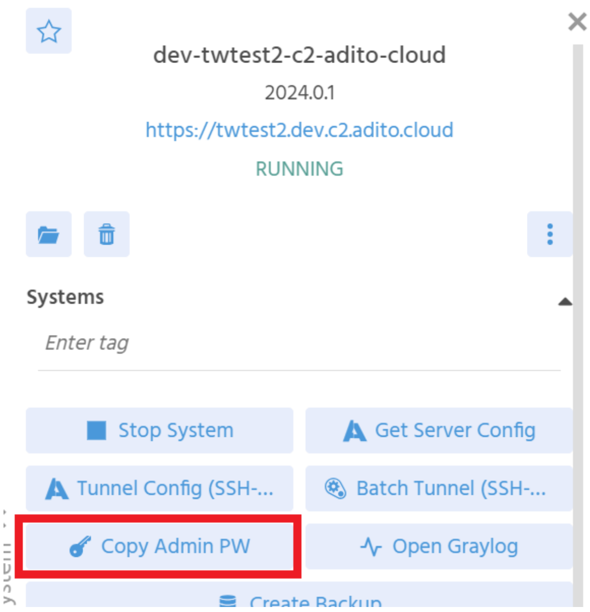
https://twtest2.dev.c2.adito.cloud

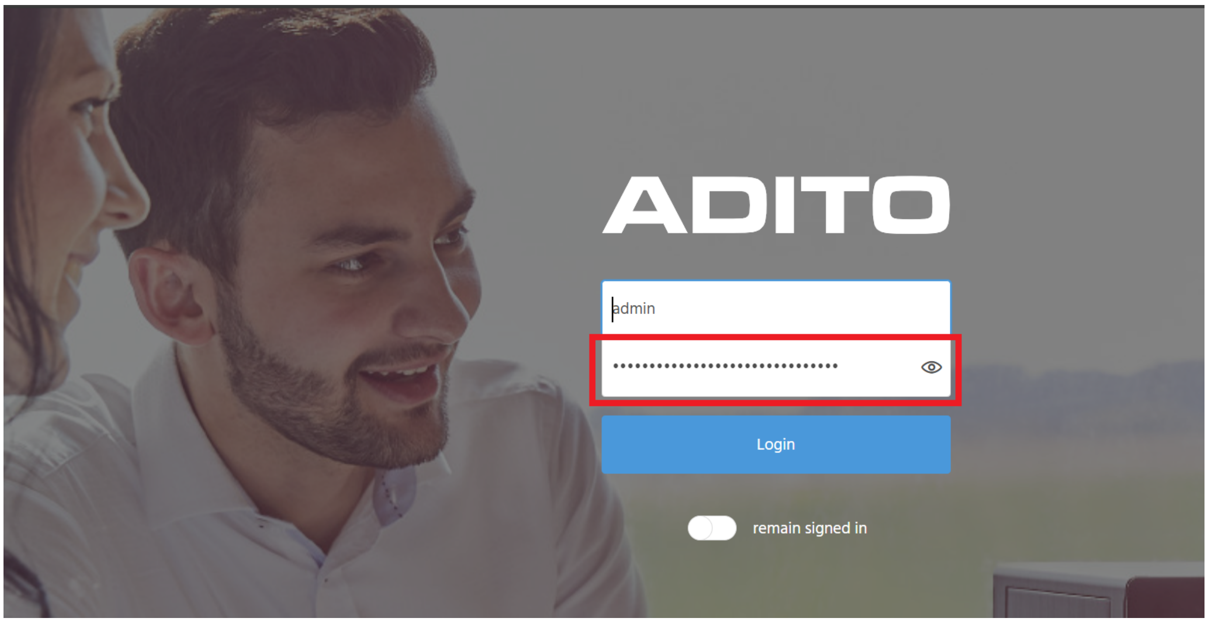
RUNNING

Systems

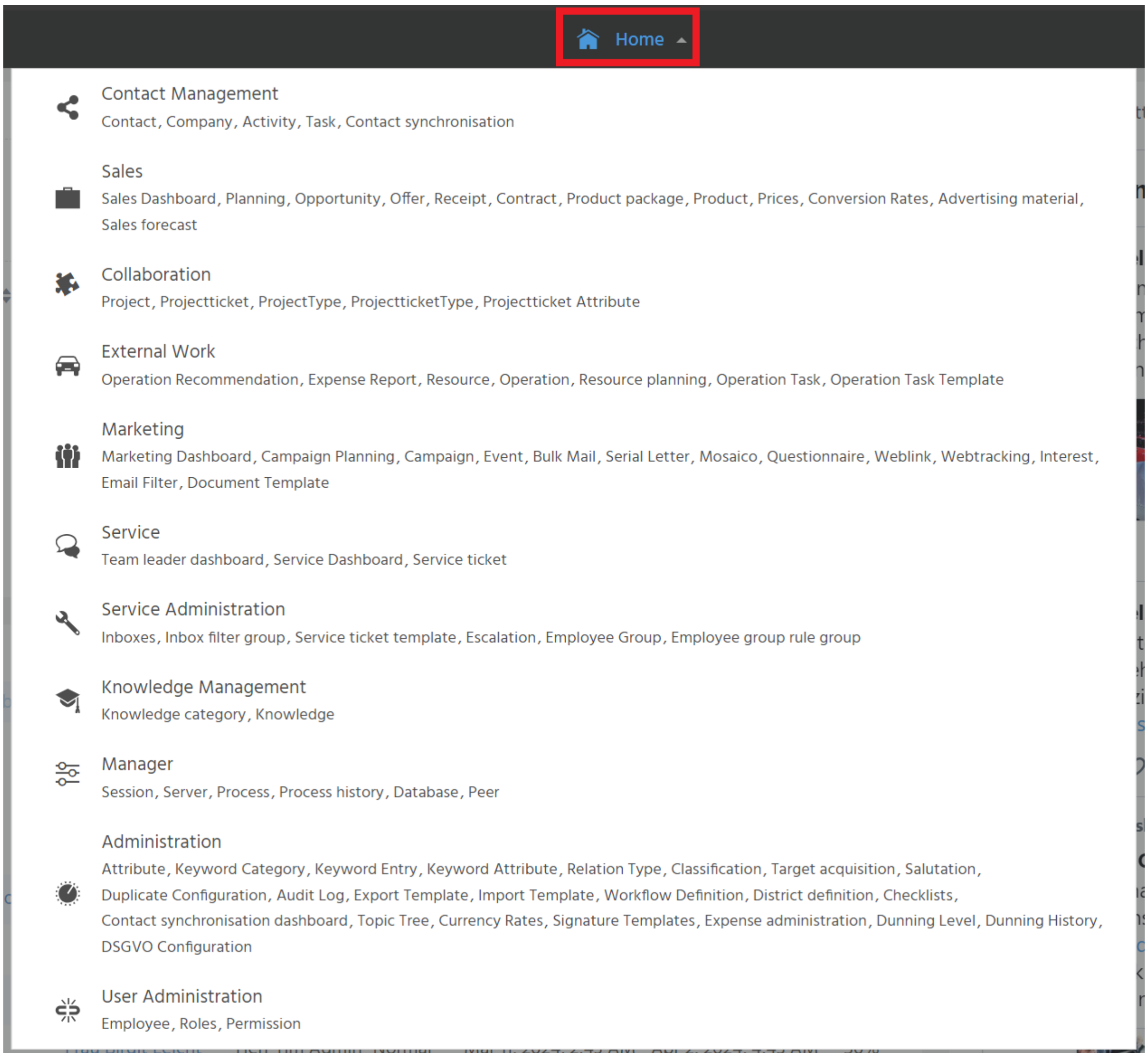
Enter tag

In the login mask, type in user name "admin" and the password available via the SSP:





After login, you are at first directed to the ADITO xRM project’s "Dashboard". You may now click on "Home", in order to show the project’s global menu - and from there, you can browse through the project’s menu groups and Contexts, in order to become familiar with the various functionalities:


Home

Contact Management
Contact, Company, Activity, Task, Contact synchronisation

Sales
Sales Dashboard, Planning, Opportunity, Offer, Receipt, Contract, Product package, Product, Prices, Conversion Rates, Advertising material, Sales forecast

Collaboration
Project, Projectticket, ProjectType, ProjectticketType, Projectticket Attribute

External Work
Operation Recommendation, Expense Report, Resource, Operation, Resource planning, Operation Task, Operation Task Template

Marketing
Marketing Dashboard, Campaign Planning, Campaign, Event, Bulk Mail, Serial Letter, Mosaico, Questionnaire, Weblink, Webtracking, Interest, Email Filter, Document Template

Service
Team leader dashboard, Service Dashboard, Service ticket

Service Administration
Inboxes, Inbox filter group, Service ticket template, Escalation, Employee Group, Employee group rule group

Knowledge Management
Knowledge category, Knowledge

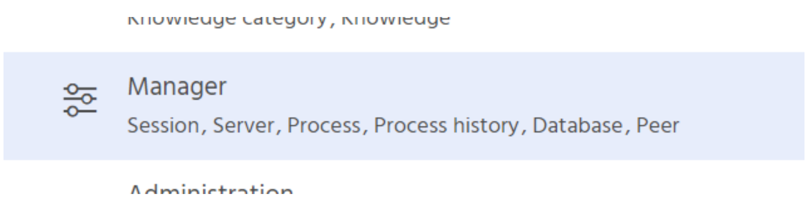
Manager
Session, Server, Process, Process history, Database, Peer

Administration
Attribute, Keyword Category, Keyword Entry, Keyword Attribute, Relation Type, Classification, Target acquisition, Salutation, Duplicate Configuration, Audit Log, Export Template, Import Template, Workflow Definition, District definition, Checklists, Contact synchronisation dashboard, Topic Tree, Currency Rates, Signature Templates, Expense administration, Dunning Level, Dunning History, DSGVO Configuration

User Administration
Employee, Roles, Permission

Via menu group "Manager", you can monitor your ADITO system, in order to, e.g.

- inspect the open sessions and, e.g., end a session, or send messages to all session-related users (Context "Session")
- monitor the server and, e.g., clear the server’s cache (Context "Server")
- view the currently running processes (Context "Process")
- view the number of open connections to the database (Context "Database")

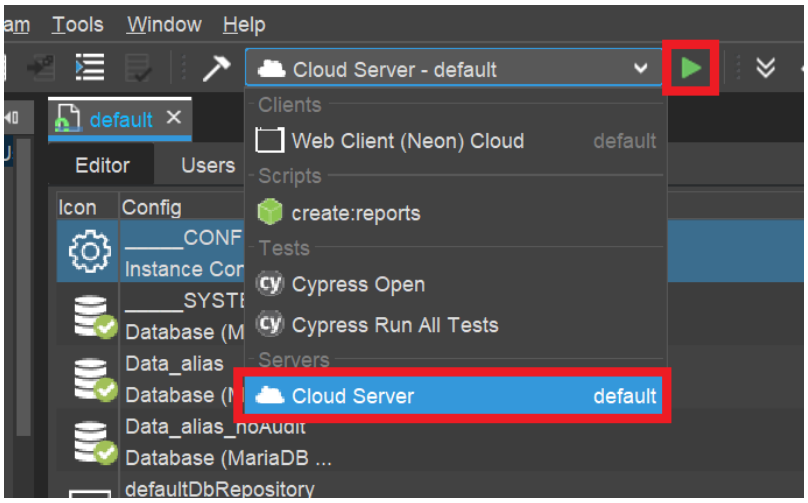

Manager
Session, Server, Process, Process history, Database, Peer

3.6. Instance configuration

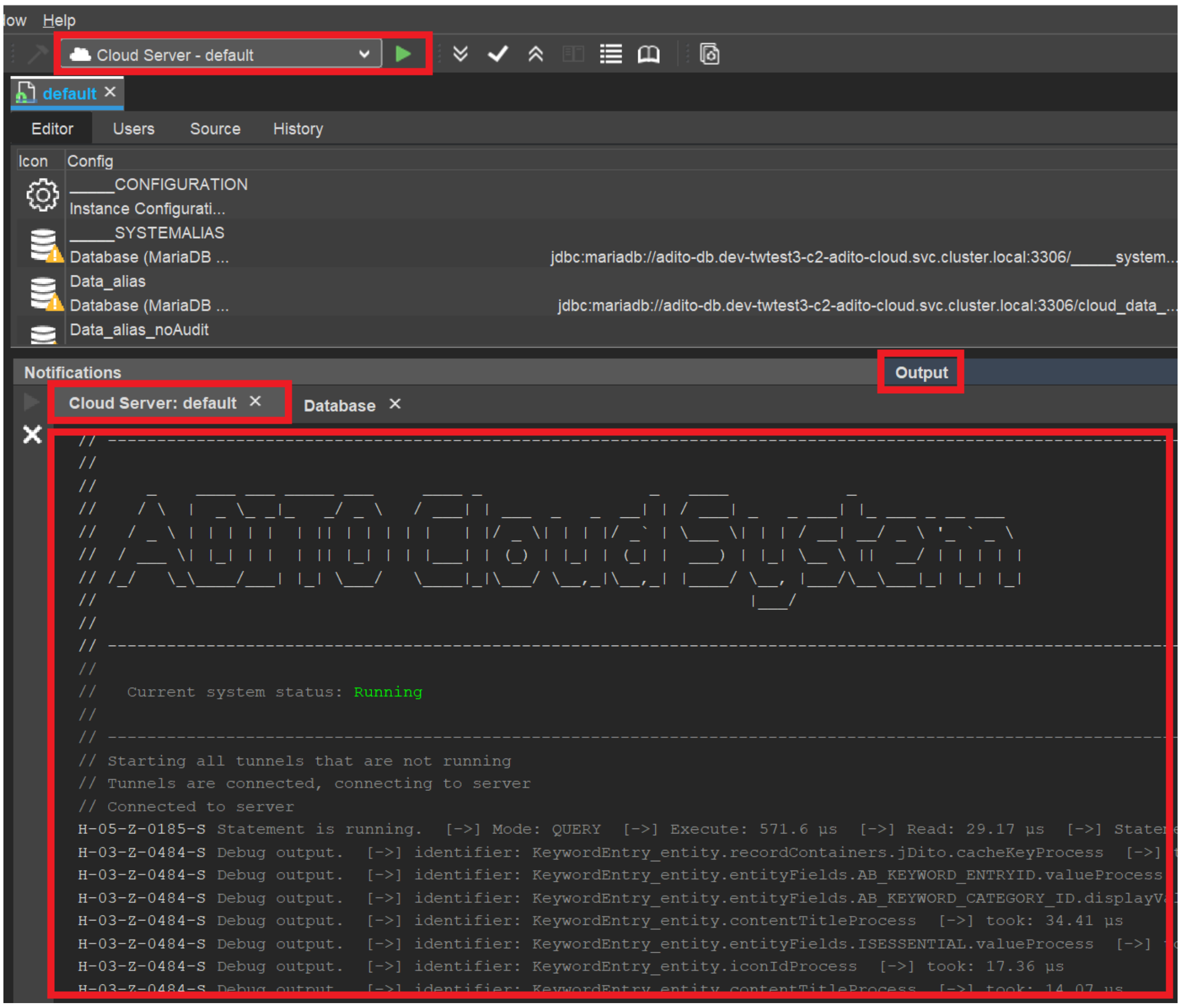
A successful database connection is also prerequisite for accessing the so-called instance configuration, holding the configuration of the ADITO system instance (e.g., what system modules should be applied, and what log levels should be effective): Double-click on system > default in the "Projects" window and then double-click on "____CONFIGURATION". This will display a configuration tree in the "Navigator" window (upper right part of the Designer). You may browse through this tree to inspect the various parts of the instance configuration.

3.7. Logging

Besides the debugger (see Designer Manual), the ADITO server's logging helps you to analyze how your system is running and what errors occur, along with the source of these errors. If the server is running, its log can be displayed via the run config named "Cloud Server - <system name>", which you can find in the combo box below the ADITO Designer's menu bar (maybe you need to scroll down in the combo box, in order to see it):



Once you have started this run config, a tab named "Output" will be displayed in the lower middle part of the Designer, which has a sub-tab named "Cloud Server: <system name>". This tab contains the log of the server:



Icon	Config	
	_____CONFIGURATION Instance Configurati...	
	_____SYSTEMALIAS Database (MariaDB ...	jdbc:mariadb://adito-db.dev-twtest3-c2-adito-cloud.svc.cluster.local:3306/_____system...
	Data_alias Database (MariaDB ...	jdbc:mariadb://adito-db.dev-twtest3-c2-adito-cloud.svc.cluster.local:3306/cloud_data_...
	Data_alias_noAudit	

```
//   Current system status: Running
//
// ------------------------------------------------------------------------------------------------
// Starting all tunnels that are not running
// Tunnels are connected, connecting to server
// Connected to server
H-05-Z-0185-S Statement is running.  [->] Mode: QUERY  [->] Execute: 571.6 µs  [->] Read: 29.17 µs  [->] State
H-03-Z-0484-S Debug output.  [->] identifier: KeywordEntry_entity.recordContainers.jDito.cacheKeyProcess  [->]
H-03-Z-0484-S Debug output.  [->] identifier: KeywordEntry_entity.entityFields.AB_KEYWORD_ENTRYID.valueProcess
H-03-Z-0484-S Debug output.  [->] identifier: KeywordEntry_entity.entityFields.AB_KEYWORD_CATEGORY_ID.displayV
H-03-Z-0484-S Debug output.  [->] identifier: KeywordEntry_entity.contentTitleProcess  [->] took: 34.41 µs
H-03-Z-0484-S Debug output.  [->] identifier: KeywordEntry_entity.entityFields.ISESSENTIAL.valueProcess  [->]
H-03-Z-0484-S Debug output.  [->] identifier: KeywordEntry_entity.iconIdProcess  [->] took: 17.36 µs
H-03-Z-0484-S Debug output.  [->] identifier: KeywordEntry_entity.contentTitleProcess  [->] took: 14.07 µs
```

3.7.1. Predefined logging

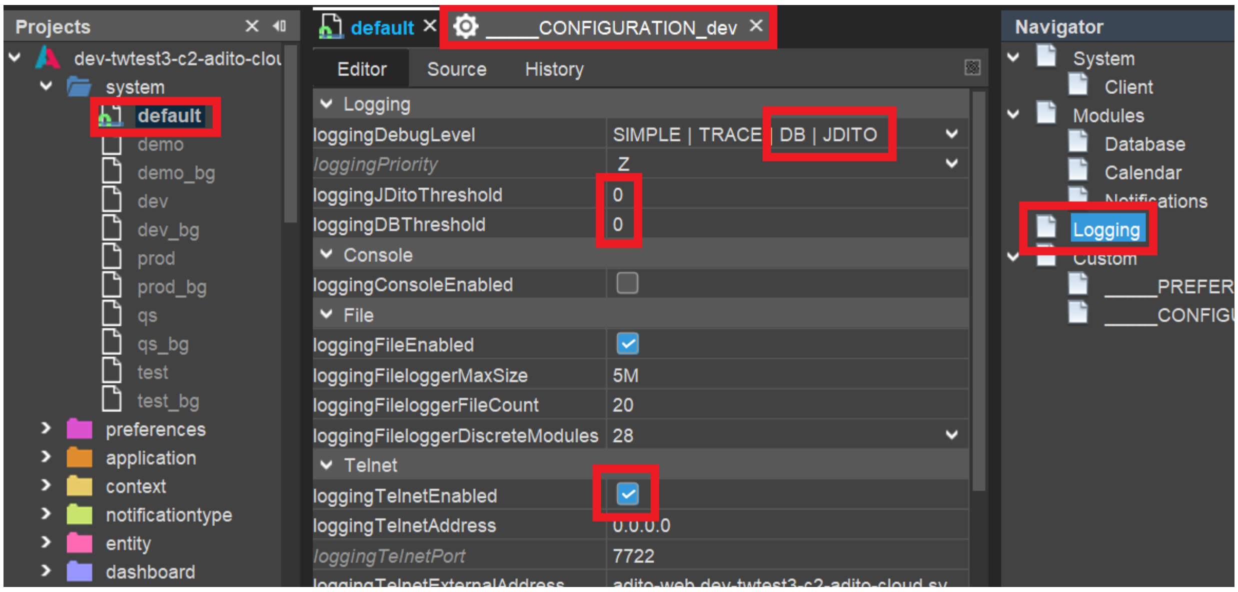
By default, only errors and a few functions are being logged. Thus, we recommend you to activate the logging for

- database access (to see, e.g., what SQL statements are executed)
- JDito processes (to see, e.g., how often the valueProcess of an EntityField is executed, which can be a hint to performance problems - see AID066 Performance Optimization)

In the instance configuration (see chapter Instance configuration), click on "Logging", which will show the logging properties in the Editor (upper middle part of the Designer). Here,

- open the combo box of property "loggingDebugLevel" and additionally check "DB" and "JDITO"
- set properties "loggingJDitoThreshold" and "loggingDBThreshold" both to "0" (zero milliseconds), meaning that **all** JDito processes and **all** database access will be logged, even very short ones.

- make sure that property "loggingTelnetEnabled" is set to true (checked)



Logging

loggingDebugLevel	SIMPLE | TRACE | DB | JDITO
loggingPriority	Z
loggingJDitoThreshold	0
loggingDBThreshold	0

Console

loggingConsoleEnabled	☐

File

loggingFileEnabled	✓
loggingFileloggerMaxSize	5M
loggingFileloggerFileCount	20
loggingFileloggerDiscreteModules	28

Telnet

loggingTelnetEnabled	✓
loggingTelnetAddress	0.0.0.0
loggingTelnetPort	7722
loggingTelnetExternalAddress	adito-web.dev-twtest3-c2-adito-cloud.sv

You can verify the success by opening any View in the client (e.g., of Context "Company", in menu group "Contact Management") and watching if the server log (in Window "Output") shows log entries for every SELECT statement and for every JDito process being executed.

> If you encounter problems, you may find a solution in chapter "Troubleshooting".

3.7.2. Customized logging

You can add further log entries to the server log, according to your requirements, by using methods of module `logging`. Simple example:

```
import { logging } from "@aditosoftware/jdito-types";

logging.log("This is the text to be logged.");
```

The JSDoc of these methods shows what additional parameters you can pass - e.g., to specifiy a certain log level. Furthermore, there are other logging methods available, e.g. `logging.logCustom` or `logging.debug`, which have further parameters that help you to optimize the log message. Find more information via the autocompletion of `logging.` or via menu Help > Show Documentation (requires plugin "Help" to be installed).

> If you want to log an object, you will get the best overview of it, if you use the JSON library to "stringify" it first:
> While `JSON.stringify(object)` returns the object’s content in one long line, you get a better result, if you set the parameters as follows

> `JSON.stringify(object, null, " ")`
> → Result (example): {
> "entity": "Person_entity",
> "object": {
> "PERSON_ID": "0a611832-9476-481e-bde5-af3c3a98f1b4",
> "CONTACTID": "a8a5f214-8165-4627-bee2-bceb3578147e",
> "FIRSTNAME": "John",
> "LASTNAME": "Smith"
> }
> }

3.7.3. Logging in "catch" section

In practice, logging is often used in the `catch` section of `try`…`catch`.
Here, a common mistake is to simply output the error itself:

Bad example of logging

```
try
{
    ()...)
}
catch(err)
{
    logging.log(err);
}
```

Often, this shows only a long stacktrace that is hard to analyze.

Instead, the log should include further information that helps to identify the problem.

Example from contentProcess of Duplicate_entity:

Good example of logging

```
(...)

    try
    {
        duplicates = duplicates.concat((new DuplicateUtils(pMappingObj)).execute());
    }
    catch (e)
    {
        logging.log(e, logging.ERROR, [
            "error while trying to load duplicates for " + vars.getString("$sys.currentcontextname") + " for user " + vars.get(
"$sys.user"),
            "Duplicate_entity.jdito.contentProcess()",
            e["rhinoException"] ? e["rhinoException"].toString() : (e.name + ": " + e.message + " " + e.stack)]);
    }

(...)
```

3.7.4. Debugging vs. temporary logging

Besides permanent logging (e.g., to log errors in the `catch` section, see above), temporary logging is sometimes used in the development process. This raises the question when to use temporary logging and when to use the ADITO Designer’s built-in debugger. Generally, the debugger provides you with a lot of options that go beyond pure logging, e.g., you can

- quickly inspect the values of *all* variables being valid at a specific code line,
- dynamically step from code line to code line, in order to watch the variables' values change
- execute functions
- define conditions when to halt at certain code lines
- manipulate variables by setting certain values
- and many more (see the chapter "Debugger" in the Designer Manual).

Therefore, the debugger should be the instrument of your choice in most cases. On the other hand, activating the debugger takes some time and decreases the system’s performance - thus, e.g., if you only want to quickly inspect the value of a specific variable, adding a simple temporary logging might be preferred:

```
logging.log("The value of myVariable at code line 173 is " + myVariable)
```

Of course, you can also combine this with a condition:

```
if (myVariable > 100) {
  logging.log("myVariable exceeds 100! Its value is " + myVariable);
}
```

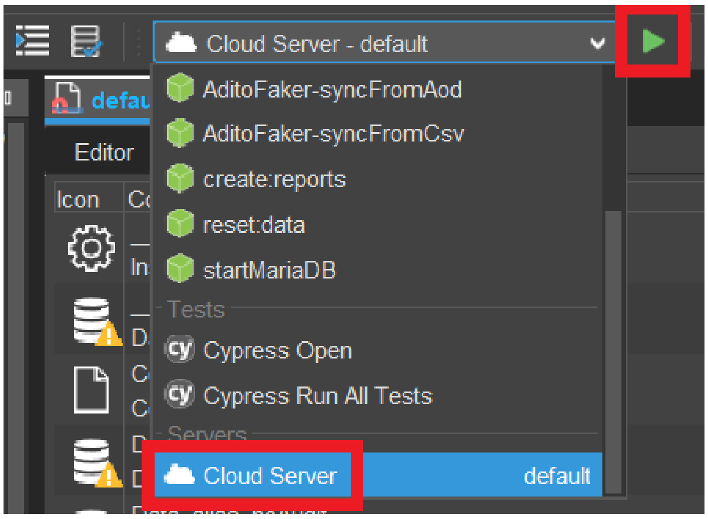
Nevertheless, a common mistake is to overload your code with temporary log entries and later forgetting to remove them again. At least, you should add an inline comment that marks the logging as temporary:

```
// TODO remove again
logging.log("My temporary logging text.");
```

In order to see the log of your cloud server in the Designer’s window "Output",

1. set the system property "loggingTelnetEnabled" to true (system > default > ____CONFIGURATION > Logging > Telnet)

2. choose "Cloud Server - <system name>" (e.g., "Cloud Server - default") from the combobox in middle of the Designer’s button bar and press the green "triangle" button to the right of it.



After a few seconds, you can read the confirmation "Connected to server" in a sub-window of the Designer’s window "Output", e.g., entitled "Cloud Server: default". In this sub-window, all further log entries will appear.

Besides this customized logging, all JDito processes and database access (SQL statements) can be logged, if you activate the corresponding log level via the system preferences:

Navigate to system > default > ____CONFIGURATION > Logging > Logging > loggingDebugLevel: Here, then check "DB" or "JDITO", respectively, and save your changes.

After a few seconds you should see various log entries in the output window, when working with your client. (Due to a bug, the logger might automatically be connected with the wrong web server pod and thus show no output. In this case, as a workaround, close and re-open the tunnels again and again, until it works. There will be a replacement for this workaround in future ADITO versions.)

4. JDito

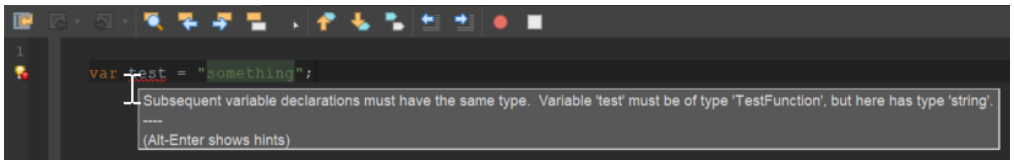
4.1. What is JDito?

JDito is the programming language used for customizing ADITO. Everything that requires more than the basic functionality offered by components is done in so-called processes, which are basically scripts written in JDito.

At its core, JDito is based on the programming language JavaScript, but it doesn’t have the DOM controlling methods like the normal JavaScript used in web development; instead, JDito extends the core of JavaScript with its own system modules that provide a big array of methods to interface with and control the environment within ADITO - e.g., executing SQL queries, interfacing with telephone systems as well as reading and returning values from/to components.

System-reserved names must not be used as names of variables. For example, a variable must not be named "result", "tools", or "test". Besides, we recommend **not** to use variables names matching names of system components, such as ADITO models or their properties ("Activity", "title", "contentType", "state", etc.). This could have unexpected side-effects. You may uses the usual prefixes, such as "my" or "a", in order to avoid these kind of problems ("myTitle", "aState", etc.).

Here is an example of how a variable irregularly named "test" is marked in the ADITO Designer’s code editor:



```
var test = "something";
```

```
Subsequent variable declarations must have the same type.  Variable 'test' must be of type 'TestFunction', but here has type 'string'.
----
(Alt-Enter shows hints)
```

4.2. How to use JDito

The lexical structure of JDito is identical to JavaScript. Basic information on JavaScript can be found online, e.g., here:
https://www.w3schools.com/js/

JDito is used in so-called processes. There are basically two kinds of processes in ADITO:

1. Component specific processes
   These are JDito scripts used for a specific purpose in a component, e.g., processes to calculate display values, font colors, data validation, etc. These processes are specified directly in the corresponding properties and are executed whenever the system needs the value of the properties.
2. Project-wide processes
   These processes are located in the "process" node of the project tree of the ADITO Designer. You find them sorted into four sub-nodes (the sorting is done according to the processes' property "variants", i.e., the sub-nodes ("folders") are virtual):
   a. authentication: All processes responsible for authentication - see the ADITO document

AID032 Authentication Methods

b. executable
   Executable processes are used to automate specific tasks and can be used manually or for regular timed tasks, e.g., nightly imports of data or mass data manipulation.

c. internal
   These are processes called by the ADITO application's core and used to define custom behaviour for specific tasks. For example, the process "autostartNeon" is called every time a user logs on to the web client. Within this process, several client-wide variables are set, like access rights.

d. library
   Processes of this kind are used to group collections of JDito function that share a common topic, like handling calendar access, writing letters, or SQL helper functions. These libraries can be imported into other processes, so you can access the functions there. There are 2 types of libraries:

   - Entity-specific libraries, including helper functions restricted to (or mainly used by) single Entitys, e.g., `Organisation_lib`.
   - multi-purpose libraries, providing functionality that is used by more than one Entity. Examples: `Neon_lib`, `Date_lib`, `Money_lib`.

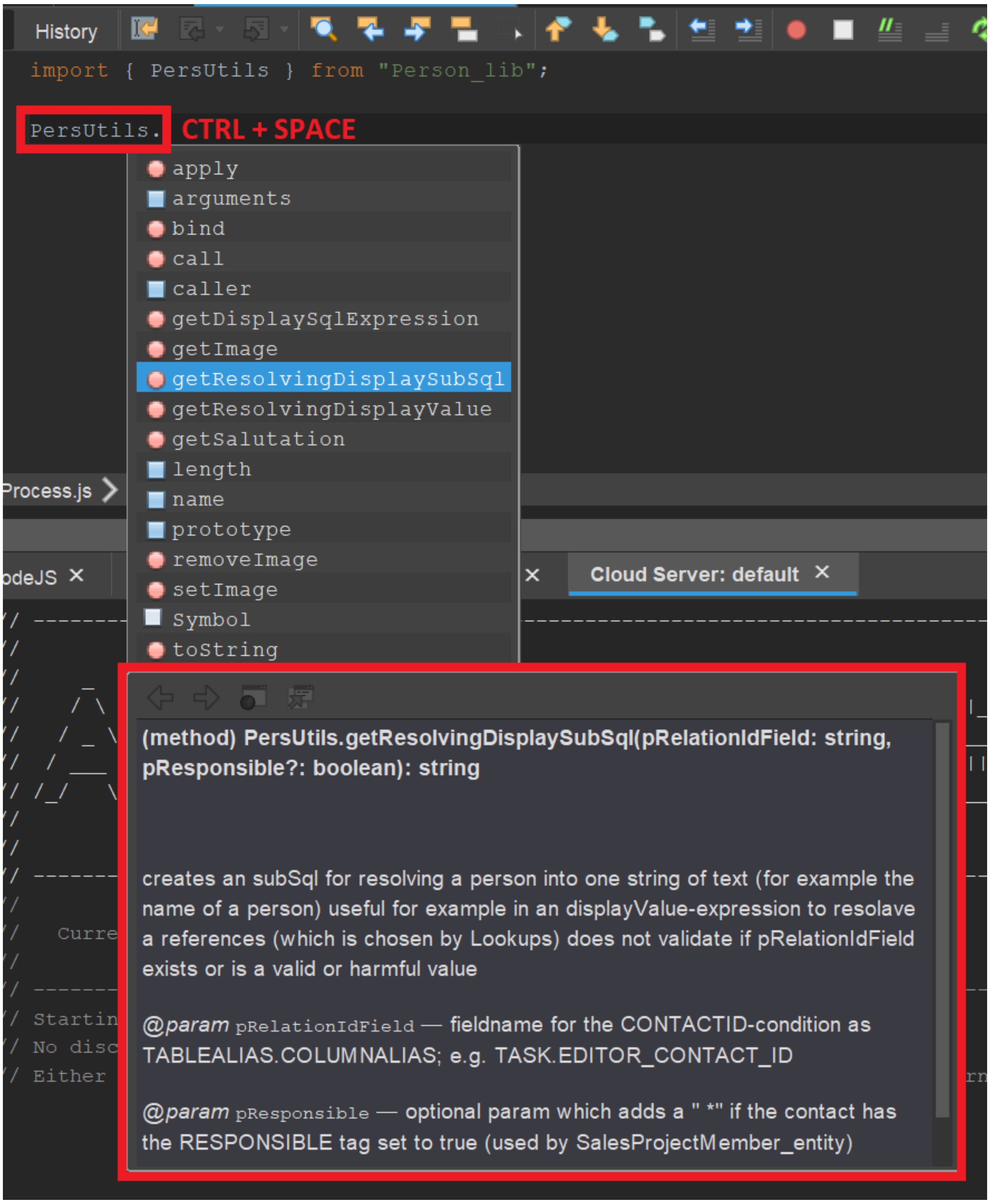
e. webservice
   These processes are designed to be used as web services. Other systems can call these to get data from ADITO, write data to ADITO, or to trigger actions within ADITO. Find more information in the ADITO document AID059 Web services with ADITO.

f. workflow
   All processes related to workflows - see the ADITO document AID110 Workflow Management

In principle, these system-wide processes can also be customized according to the project's requirements. However, this can lead to update/merge problems whenever ADITO releases a new xRM version. Therefore, we recommend you to create new processes for any customized functionality, e.g., KeywordRegistry_custom or MyNewContext_lib.

4.3. Further information

Further information on JDito functionality is available as JSDoc, accessible via the programming help function while coding. For example, if you have imported the library "Person_lib" (`import { PersUtils } from "Person_lib";`), you can view a list of all functions provided by class "PersUtils" simply by typing `PersUtils.` and then CTRL+SPACE. (If you do this for the first time, you need to wait a few minutes, until the ADITO Designer has completed the task "Initializing JS features", see notification in the bottom line of the Designer). Through this function list, you can navigate with the arrow keys: Whenever a function is marked, you can read its JSDoc below, structured in Summary (basic description of the function), Parameters (description of the function’s parameters), and Returns (information about the function’s return value).



```
import { PersUtils } from "Person_lib";

PersUtils.
```

```
apply
arguments
bind
call
caller
getDisplaySqlExpression
getImage
getResolvingDisplaySubSql
getResolvingDisplayValue
getSalutation
length
name
prototype
removeImage
setImage
Symbol
toString
```

(method) PersUtils.getResolvingDisplaySubSql(pRelationIdField: string, pResponsible?: boolean): string

creates an subSql for resolving a person into one string of text (for example the name of a person) useful for example in an displayValue-expression to resolave a references (which is chosen by Lookups) does not validate if pRelationIdField exists or is a valid or harmful value

@param `pRelationIdField` — fieldname for the CONTACTID-condition as TABLEALIAS.COLUMNALIAS; e.g. TASK.EDITOR_CONTACT_ID

@param `pResponsible` — optional param which adds a " *" if the contact has the RESPONSIBLE tag set to true (used by SalesProjectMember_entity)

Figure 4. Example of the JSDoc of method getResolvingDisplaySubSql of class PersUtils

A glossary giving information about JDito system modules and JDito system variables is available in appendix JDito system modules and variables.

Information about how to use XML in JDito is available in appendix XML in JDito.

5. Core tables of the xRM project

As already mentioned, we will build our ADITO example application based on the xRM project. This is a comprehensive project, which already includes several Entities, e.g., for managing contact persons, companies, activities, products, offers, and administrative tasks.

Every database table has got a primary key column, which is named <table name>ID (e.g., ORGANISATIONID), according to ADITO's spelling guidelines (see ADITO Information Document AID001, chapter "Spelling & Wording" > "ADITO models"). Whenever a primary key column is referenced in other tables ("foreign key"), it is named <table name>_ID (e.g., ORGANISATION_ID). This simplifies the orientation in the ADITO data model, as you can quickly recognize the relations between specific tables. In most cases, the primary key value is a UID (a 36-digit universally unique identifier that is generated using random numbers).

In the xRM project, database columns holding foreign keys usually do not have a foreign key constraint. This has multiple reasons (e.g., it simplifies the task to drop and re-create database tables, and it grants more flexibility when creating interdependent datasets), but nevertheless it's up to your own programming style whether or not you do it alike when adding your custom database tables.

As an xRM system particularly focusses on relationships between persons or organisations, the core database tables of the ADITO xRM project are (cf. illustration below):

- PERSON: Holds data of persons, like name, date of birth, etc.
- ORGANISATION: Holds data of organisations, which are, in most cases, companies: Name, type, info, etc.
- CONTACT:
  - Connects persons and organisations, as well as corresponding addresses, communication data, and activities. In ADITO, a PERSON or ORGANISATION dataset never exists alone, but it is always connected to at least one CONTACT dataset. This is, because a person is usually seen as a "person in an organisation". And an organisation usually features one or more persons working in it. The connection is realized via CONTACT's fields PERSON_ID and ORGANISATION_ID.
  - Also for private persons, a related CONTACT dataset is created - in this case, the field referencing the organisation (ORGANISATION_ID) has the value 0 (not "null"!).
  - Likewise, an organisation without relation to a person is nevertheless represented also as CONTACT dataset - in this case, the field referencing the person (PERSON_ID) is null (not

"0"!).

    - **Whenever we reference a person or an organisation in ADITO logic, we always use the corresponding CONTACT_ID, not the PERSON_ID or ORGANISATION_ID.** However, of course, the tables PERSON and ORGANISATION may be joined in SQL statements, e.g., to retrieve the name of a person/organisation.
    - The standard address of a contact (cf. ADITO User’s Manual) is referenced in column CONTACT.ADDRESS_ID. This is no mandatory field, but as soon as at least 1 address of a contact exists, one of these adresses must be assigned as standard address.

- ADDRESS: Holds address data, along with a CONTACT_ID, referencing the contact (person or organisation) to which the address belongs.
- COMMUNICATION: Holds information about communication ways (telephone number, email address, etc.), along with a CONTACT_ID, referencing the contact (person or organisation) to which the communication data belongs.
- ACTIVITY: Holds information about activities. This term summarizes information about all kinds of events belonging to specific ADITO Contexts, e.g., a meeting or a telephone call and its result. As one contact (person or organisation) can be related to multiple activities, table ACTIVITYLINK connects the tables ACTIVITY and CONTACT, via its columns ACTIVITY_ID and OBJECT_ROWID. The latter can hold a CONTACTID, but is named universally, as activities can also refer to other ADITO Contexts, such as "Opportunity" or "Contract".

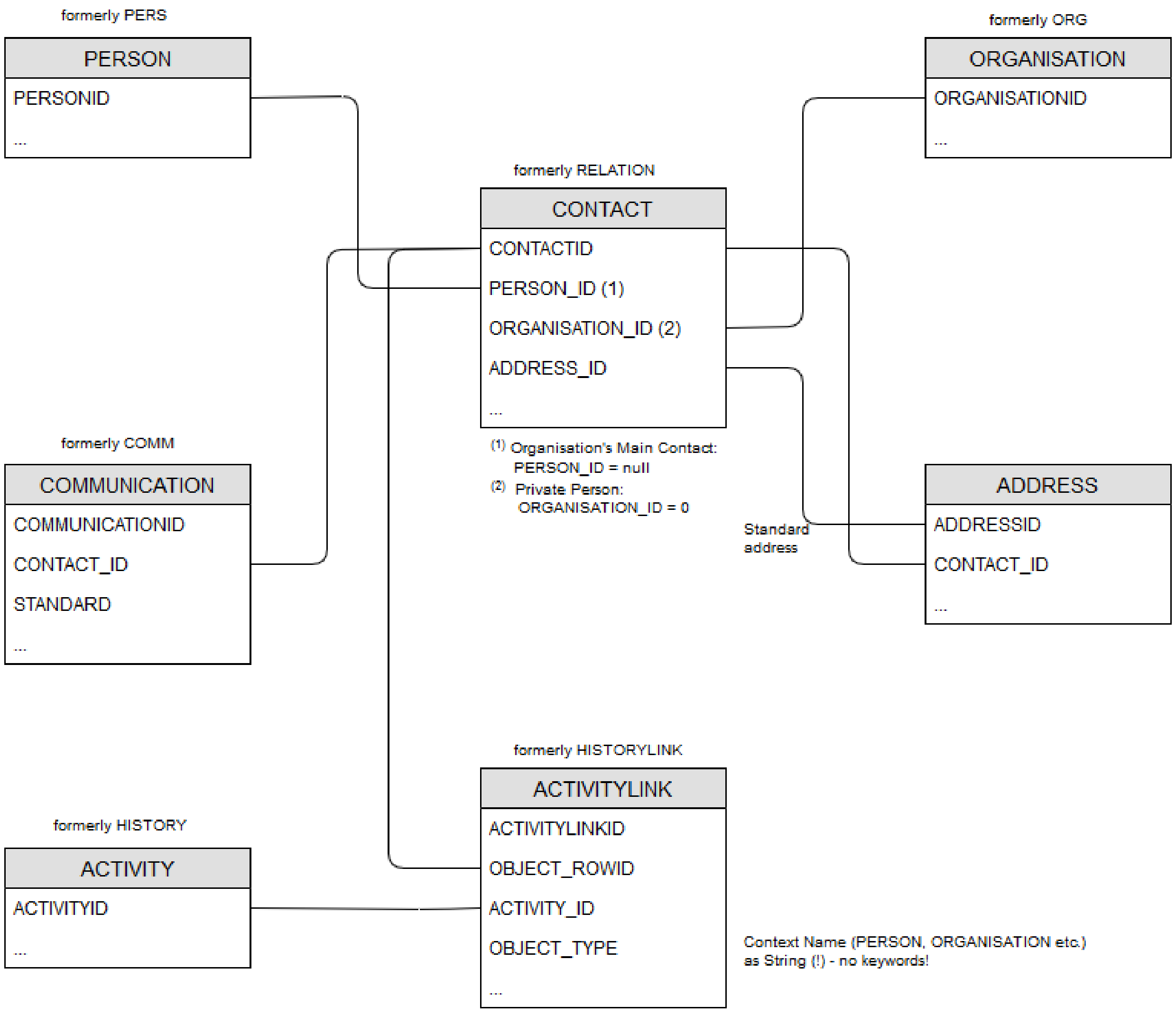


Figure 5. ER diagram of the ADITO xRM project’s core tables (along with their former names, until ADITO 5)

However, unlike normally, the ADITO Entities corresponding to these core database tables are not 1:1 representations, although they are named similarly:

- Person_entity represents contact persons and is related to both database tables PERSON and CONTACT. In the web client, the data held by Person_entity appears under the title "Contact".
- Organisation_entity represents organisations and is related to database tables ORGANISATION and CONTACT. In the web client, the data held by Organisation_entity appears under the title "Company".
- Contact_entity represents a special case and is not a 1:1 representation of the data of database table CONTACT. The data held by Contact_entity does not appear 1:1 in the web client. Do not confuse it with the data held by Person_entity, which is titled "Contact" in the web client. In this manual, we can ignore Contact_entity, until further notice.

- AnyContact_entity is a kind of mixture between Person_entity and Organisation_entity: Both persons (private and company-related) and organisations (without persons assigned) are displayed. It also represents a special case and can be ignored in this manual, until further notice.

6. Modelling the data structure

The first step of the ADITO customizing work basically requires nothing more than a pencil and a sheet of paper:

Collect a list of all data you want to manage via ADITO, and then structure it carefully into logical units with certain features.

Do not underestimate this step: The more complete and the better structured this collection is, the more efficient the following development process will be. Carefully consider structural principles like normalization, consistency, and avoidance of redundancy. Later extensions or modification of data structure and logic are, of course, possible with ADITO, but - as in every engineering process - they require additional effort that is probably larger than if you had considered it more carefully right from the beginning. To put it bluntly, you should not touch the ADITO Designer unless you are very sure that your data feature collection is both complete and optimally structured.

In this manual, we will use the administration of a company car pool as example. Our list of data features will therefore contain, e.g., the basic features of the car (manufacturer, type, manufacturing date, color…), the personal data of the drivers (their name, the IDs of their driving licenses, etc.), and reservation data (start date, end date, corresponding car, corresponding driver, etc.).

As soon as we are sure that our data list is complete, we cluster the data into business objects, each having several features. If a feature is related to another feature, or if it can be deduced from other features or somehow be calculated, we mark it accordingly.

Every business object features am ID, in order to ensure a unique identification.

In our car pool example, the business objects could be modelled like this:

CAR:

- ID
- Manufacturer
- Type
- Color
- Date of Manufacture
- Picture

- Price
- Currency
- License plate number
- Mileage (calculated from car reservation)
- Value (calculated from mileage)
- Availability ("available"/"lent", dependent on car reservation)
- Damages (calculated from reservations)

CAR DRIVER:

- Car driver ID
- Contact ID (related to xRM Entity "Person_entity")
- Last name (retrieved via Contact ID)
- First name (retrieved via Contact ID)
- Age (calculated from date of birth via Contact ID)
- Number of driving license
- issue date of driving license
- Driving experience (calculated from issue date)
- Sum of parking ticket fines (calculated from car reservations)
- Sum of speeding fines (calculated from car reservations)

CAR RESERVATION:

- Car reservation ID
- Car driver ID (related to business object "Car driver")
- Car ID (related to business object "Car")
- Start date
- End date
- Mileage at start (calculated from car reservation)
- Mileage at return
- Damage

- Parking ticket fine
- Speeding fine
- Currency

7. Creating Entities

This chapter explains how to create Entities and their EntityFields manually, step-by-step. The connection to the database will be done subsequently. This approach takes some time, but you will learn to understand the details. However, ADITO includes automatisms called "Blueprints" that simplify the creation of Contexts, Entities, and Views. Find more information in chapter Blueprints.

Once we have designed our business objects, we are ready to start the development of the actual ADITO application. Our central development tool is the ADITO Designer, which you can download as compressed (zip) file from area "Aktuelle Releases" (current releases) in the customer area of ADITO’s website.

Now, start the ADITO Designer and open your project.

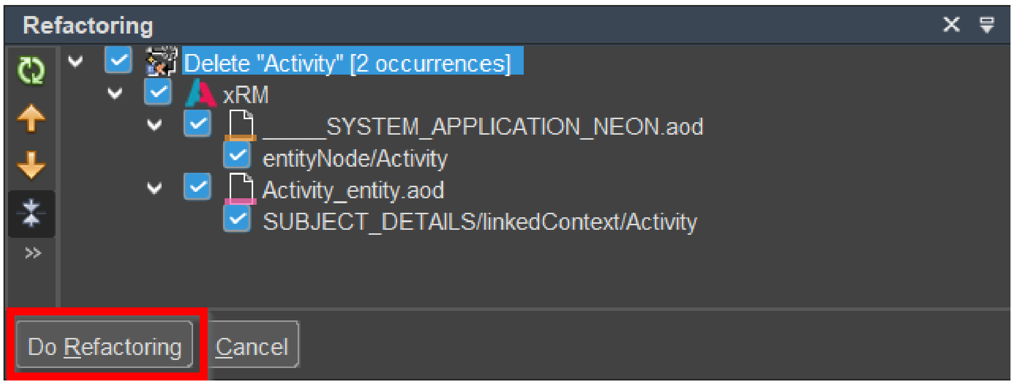
In ADITO, all business objects are represented by so-called Entities. They are the basic elements to model the data’s structure and type. For every business object, we create one Entity, and for every feature of the business object, we create one EntityField. As the spelling of an Entity name follows the convention "<Name in camel case>_entity", we call our Entities

- Car_entity
- CarDriver_entity
- CarReservation_entity

(By letting all names start with "Car", we make them being displayed close together in various ADITO folders, because of the alphabetical sorting.)

To create new Entities, we navigate to node "entity" in the project tree (see "Projects" window to the left) and call option "New" in the context menu of "entity". In the model creation dialog, we enter the name of the Entity and leave "entity" selected as type. This step must be repeated until all Entities are created.

In case you mistype a name or want to delete an element of the project, please note: In some cases, particularly when deleting or renaming, a tab "Refactoring" will appear in the lower middle part of the Designer (it can very easily be overlooked!), which requires you to confirm the refactoring by clicking on button "Do Refactoring". Here you see all models affected by the refactoring, and on demand, you can uncheck part of them (not recommended!). If you miss to react to the refactoring prompt (or repeat the action that caused it) and continue working, your XML project source code might become confused and you will have to repair it manually.


Refactoring

Delete "Activity" [2 occurrences]
xRM
____SYSTEM_APPLICATION_NEON.aod
entityNode/Activity
Activity_entity.aod
SUBJECT_DETAILS/linkedContext/Activity

Figure 6. Example of content of tab "Refactoring"

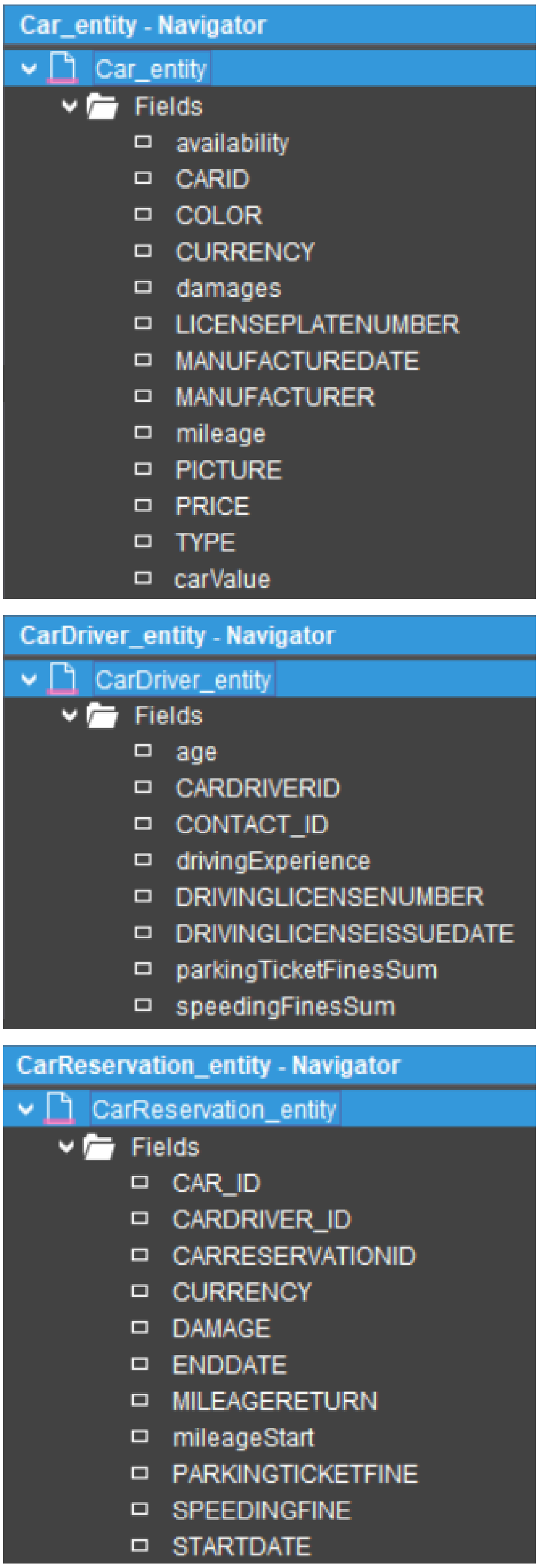
To create new EntityFields, we double-click on an Entity name (under node "entity" in the "Projects" window) and then call option "New Field" in the context menu of the Entity’s name shown in the Navigator (!) window. (This is the window in the upper right part of the Designer.) According to ADITO’s spelling guidelines (see ADITO Information Document AID001, chapter "Spelling & Wording" > "ADITO models")

- the names of EntityFields that directly refer to the value of a database column, are written in uppercase letters (same name as the corresponding database column);
- all other EntityField names (calculated fields, etc.) are written in camelCase, starting with a lowercase letter.

In practice, the names of Entities, Contexts, Views etc. include a suitable project-related prefix, e.g., "MyProject_CarReservation_entity". This helps you to easily distinguish original ADITO models (= models of the xRM project) from your own ADITO models that you have created for your customized project. However, in this manual, we do not use a project prefix, for purposes of simplification.

As the correct spelling of the EntityFields' names is essential for the function of the following code examples, you can find tables with the names (and contentTypes/data types - see later chapters) of all car pool related EntityFields also in appendix Car pool example: EntityFields - ready for "copy & paste".

After creating all Entities with all of their fields, the Entities should appear in the Navigator window as follows:


Car_entity - Navigator

- Car_entity
  - Fields
    - availability
    - CARID
    - COLOR
    - CURRENCY
    - damages
    - LICENSEPLATENUMBER
    - MANUFACTUREDATE
    - MANUFACTURER
    - mileage
    - PICTURE
    - PRICE
    - TYPE
    - carValue

CarDriver_entity - Navigator

- CarDriver_entity
  - Fields
    - age
    - CARDRIVERID
    - CONTACT_ID
    - drivingExperience
    - DRIVINGLICENSENUMBER
    - DRIVINGLICENSEISSUEDATE
    - parkingTicketFinesSum
    - speedingFinesSum

CarReservation_entity - Navigator

- CarReservation_entity
  - Fields
    - CAR_ID
    - CARDRIVER_ID
    - CARRESERVATIONID
    - CURRENCY
    - DAMAGE
    - ENDDATE
    - MILEAGERETURN
    - mileageStart
    - PARKINGTICKETFINE
    - SPEEDINGFINE
    - STARTDATE

Figure 7. The carpool-related Entities and their fields

Make sure that you named Entities and Entity fields exactly as shown above. Otherwise, some code snippets of this manual might not work.

Every Entity and every EntityField (as well as many other elements of the ADITO project) have several features, which are called properties. Generally, there are two ways how properties can be displayed and edited:

- Click on an Entity name (in the "Projects" windows or in the "Navigator" window) or on a field name (in the "Navigator" window): The properties are visible and (partly) editable in the "Properties" window, which is by default located in the lower left part of the ADITO Designer.
- Right-click on an Entity name (in the "Projects" windows or in the "Navigator" window) or on a field name (in the "Navigator" window): Choose option "Properties" in the context menu. Then, a popup window will appear, showing the same content as the "Properties" window mentioned above.

In both cases, the lower part of the property field is a text area showing a short documentation: If you click on a property, a summary of the property's purpose or usage will be displayed.

7.1. Configuring Entities

For every Entity, we set the following properties:

- `title`: A general title summarizing the content of the whole Entity. In many cases, this title is simply the first part of the Entity's name as specified when creating it: Car, Car driver, and Car reservation. The title will be shown in various parts of the client, e.g., as headline on the top of a Context.
- `titlePlural`: The plural form of the title (see above); will be used, e.g.,
  - in the FilterView of the Entity's Context: before the number of datasets
  - in the MainView including the Entity's data as reference: as tab title
- `contentTitleProcess`: A piece of code for retrieving a suitable title summarizing the content of a single dataset. This "contentTitle" will be used in different parts of the ADITO logic - in particular, on the top of the MainView and for creating the list items of a combo box of an EntityField that gets its values via a Consumer (this term will be explained further below).

  In the MainView, the contentTitle will only be visible, if at least one ViewTemplate is assigned to the "Detail" area of the MainView's MasterDetailLayout. (This will be done in a later step of the car pool example, along with an explanation of these terms.)

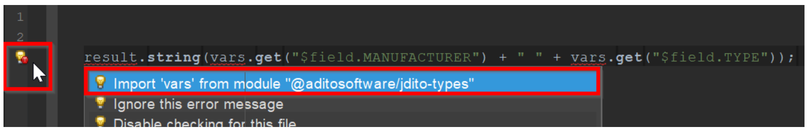
Usually, a contentTitle consists of the values of a column or a combination of multiple columns. In our carpool example, we define the contentTitleProcess of our Entities as follows:

- Car_entity: We use a combination of the columns MANUFACTURER and TYPE, separated by a whitespace.

  Car_entity.contentTitleProcess

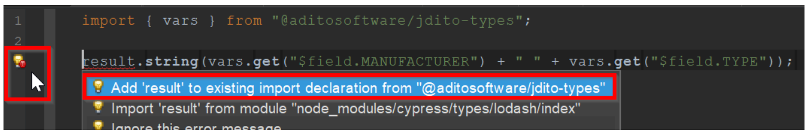
  ```
  import { result, vars } from "@aditosoftware/jdito-types";

  result.string(vars.get("$field.MANUFACTURER") + " " + vars.get("$field.TYPE"));
  ```

  In ADITO, the leading code lines, which start with `import`, make all required system modules (e.g. "result" or "vars") available for being called in the code (e.g., via `result.string` or `vars.get`). For reasons of simplification, most of the following code fragments in this manual will not include these "import" lines. You can easily add them, if you save your code and wait a few seconds, until a "lightbulb" icon appears to the left of the respective code line. (At first, the process "Initializing JS features" will automatically run - see the waiting bar in the lower part of the Designer.) Then click on this light bulb and choose "Import '…' from module @aditosoftware…"" (or "Add…", respectively), which automatically adds/extends the required `import` line.

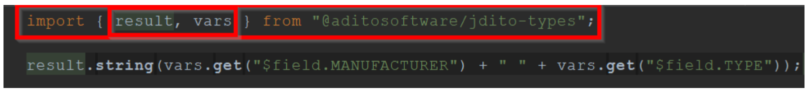
  ```
  result.string(vars.get("$field.MANUFACTURER") + " " + vars.get("$field.TYPE"));
  ```

  Import 'vars' from module "@aditosoftware/jdito-types"
  Ignore this error message
  Disable checking for this file

  ```
  import { vars } from "@aditosoftware/jdito-types";

  result.string(vars.get("$field.MANUFACTURER") + " " + vars.get("$field.TYPE"));
  ```

  Add 'result' to existing import declaration from "@aditosoftware/jdito-types"
  Import 'result' from module "node_modules/cypress/types/lodash/index"
  Ignore this error message

  ```
  import { result, vars } from "@aditosoftware/jdito-types";

  result.string(vars.get("$field.MANUFACTURER") + " " + vars.get("$field.TYPE"));
  ```

- CarDriver_entity: We display the driver’s name in cleartext, which must be retrieved from table PERSON, using a prepared SQL statement via the class SqlBuilder. If you are interested to know what SQL code this helper function returns, please refer to appendix Database Access, chapter "SQL Helper Functions".

CarDriver_entity.contentTitleProcess

```
var contactId = vars.get("$field.CONTACT_ID");

if (contactId) {

    var displayData = newSelect("SALUTATION, FIRSTNAME, LASTNAME")
    .from("PERSON")
    .join("CONTACT","CONTACT.PERSON_ID = PERSON.PERSONID")
    .where("CONTACT.CONTACTID", contactId)
    .arrayRow();

    if(displayData) {

      var salutation = displayData[0];
      var firstname = displayData[1];
      var lastname = displayData[2];

      result.string(salutation + " " + firstname + " " + lastname);
    }
}
```

- CarReservation_entity: In addition to the CARRESERVATIONID, we also display the driver's name in cleartext, which must be retrieved from table PERSON, using a prepared SQL statement with helper functions. If you are interested to know what SQL code this helper function returns, please refer to appendix Database Access, chapter "SQL Helper Functions".

CarReservation_entity.contentTitleProcess

```
var carReservationId = vars.get("$field.CARRESERVATIONID");
var carDriverId = vars.get("$field.CARDRIVER_ID");

if (carReservationId && carDriverId) {

    var displayData = newSelect("FIRSTNAME, LASTNAME")
    .from("PERSON")
    .join("CONTACT", "CONTACT.PERSON_ID = PERSON.PERSONID")
    .join("CARDRIVER", "CARDRIVER.CONTACT_ID = CONTACT.CONTACTID")
    .where("CARDRIVER.CARDRIVERID", carDriverId)
    .arrayRow();

    if(displayData) {

      var firstname = displayData[0];
      var lastname = displayData[1];

      result.string(carReservationId + ", for " + firstname + " " + lastname);
    }
}
```

In the Editor window (upper middle part of the Designer), you can scale (zoom) the font size of the code lines up and down by pressing the mouse wheel and then turning it back and forth.

Loading and writing datasets via `SqlBuilder` (or via the older methods `db.xxx`) ignores the permissions (access rights) configured by the client administrator! To load or write data respecting these permissions,

- set property "usePermissions" of the respective Entity/EntityFields to "true" (checkbox checked) **and**
- use the functionality of "LoadEntity" and "Write Entity" instead - see appendix LoadEntity and WriteEntity. However, please note: Using "LoadEntity" causes a considerable overhead when executed often, as all required Entity processes are executed as well. The contentTitleProcess is a very good example in which it will cause performance issues, because it is executed for every record loaded and in turn executes all processes of each record it loads.
- For further information on setting permissions please refer to the ADITO documentation for client administrators.

You should use the same loading principle for all other processes quoted in this manual, whenever you want access rights to be respected. For purposes of simplification, the further code examples do not respect access rights.

7.2. Configuring EntityFields

For all EntityFields, we set the property `title`: Insert a term or short text suitable for this EntityField. This text will be shown in various Views in the client (e.g., as column title or field label). Furthermore, in the EditView (create mask), the title will also be shown as placeholder in an input field, as long as it is empty. If you want to use a specific placeholder instead, you can set this in property `placeholder`.

Some EntityFields require the setting of specific properties:

- contentType: You can find tables with the content types of all car pool related EntityFields in the appendix Car pool example: EntityFields. The default state of this property is "TEXT", as this is the most common content type. Still it is recommended not to leave the default state, but to set "TEXT" manually (simply change the value to any other content type and then set it back to "TEXT" - then the default state is left, which you can recognize by the white font color of the property name - see Designer Manual.) This makes sure that the value of property contentType will remain "TEXT", even if the default state is changed to any other content type in future ADITO versions. In appendix Content types you can find an overview of all available content types and their features.

- As the content of the field PICTURE (of Car_entity) is an image, select "IMAGE" in the combo box of the field’s property `contentType`. (NOTE: To avoid long loading and rendering times, images in ADITO should not exceed a certain limit as for resolution and size. Find more information in document AID066 Performance Optimization.)
- For all fields holding a date
    - set property `contentType` to "DATE"
    - set property `resolution` to a suitable resolution type (e.g., "DAY"). This value determines how precise the date is saved (and displayed, if property "outputFormat" is not set).
    - optionally, change the default date format by inserting a format pattern (e.g., "yyyy-MM-dd" or "EEE, d MMM yyyy HH:mm:ss") in the property `outputFormat` or `inputFormat`. But CAUTION:
        - Property "resolution" is still valid, as far as the saving of the data is concerned, i.e., the data might, e.g., be stored less precise than entered.
        - If property outputFormat or inputFormat is set, the format will be used for all languages. This means that, e.g., users from the USA will see the German date format.
- For all fields that must not be let empty, set property `mandatory` to true (usually, this should be true at least for all fields that correspond to a "not null" database column, see below).
- For all fields that should act as grouping criteria (see FilterView, section "Grouping > Group by", in combination with ViewTemplate type "TreeTable"), set property `groupable` to true - e.g., for MANUFACTURER, TYPE, or COLOR.
- For all fields that must not be edited (in particular, calculated fields and primary key fields), set property `state` to READONLY (copy of value possible) or DISABLED (copy of value not possible, font grayed).

The property `description`, which is included in many ADITO models (Entity, EntityField, View, etc.), has no effect in the client and can be ignored. (This might change in future ADITO versions.) You can add descriptive content in other properties:

- `documentation`: Here, you can describe whatever you want other programmers to know about the respective model. This text has no effect in the client.
- `tooltip`: The text entered here will appear in the client, whenever you

hover with the mouse pointer over the name/title of the respective model. (This is not implemented for all models.)

- `placeholder`: This is a property of an EntityField. The text entered here will appear in the client in the respective input fields of the EditView (create mask), as long as nothing has been entered. As soon as you start to fill in the input field, the placeholder text will disappear. If this property is not set, the value of property `title` will be used as default.

- There are properties restricting the length of an EntityField: maxFieldSize, maxIntegerDigits, and maxFractionDigits. These 3 properties are not always present, but they differ according to the EntityField’s contentType (see below). If property contentTypeProcess is set, all 3 properties are shown, but only those properties are evaluated that are suiting the contentType given in the result of the contentTypeProcess.
  If the specified length is exceeded, the "save" button is disabled, and beside it, a corresponding message is shown. On the server side (e.g., when using WriteEntity) an exception is thrown, including information about the EntityFields causing the error.
  By default, the properties have the value "<unlimited>" (= no restriction). The method project.getEntityStructure also includes the 3 properties.
  Special cases can be handled via the onValidationProcess.
  Further information about the length-restricting properties:
  - maxFieldSize: This property is available for EntityFields of contentType TEXT, LONG_TEXT, HTML, TELEPHONE, EMAIL, LINK, and PASSWORD. It limits the number of characters that can be entered.
  - maxIntegerDigits: This property is only available for EntityFields of contentType NUMBER. It limits the integer digits of a number (= the number of numbers before the decimal point).
  - maxFractionDigits: This property is only available for EntityFields of contentType NUMBER. It limits the number of decimal places (= the number of numbers after the decimal point). If required, you can enter "0" here, in order to limit the input to integers; however, in this case, you should also set a suitable input format, that restricts the input accordingly.
  - Please note that for EntityFields of contentType NUMBER, there are also the properties maxValue(Process) and minValue(Process) available.

Mass edit support:
The ADITO Designer includes a function that enables you to set the same value for the same property of multiple objects at once. This is especially helpful when configuring the properties of EntityFields: Just mark 2 or more fields, consequently

clicking on them while the "CTRL" key is being held. Then, the title bar of the property window changes to "Multiple Objects - Properties". If you then set a property value, e.g. "tooltip" or "mandatory", it will be set for all marked fields simultaneously.
If you have marked multiple EntityFields, you can read "<Different Value>" for all properties whose values are different. If you set a value here, all existing values are overwritten.
This mass edit function is not available for properties whose values are set via a tab in the Editor area (upper middle part of the Designer), i.e., it works, e.g., for property "tooltip", but not for "tooltipProcess".

Be aware that all fields carrying data are subject to your project’s access rights management as well as aspects of data security. Therefore, make sure that

- your ADITO client administrator knows about **every** EntityField (even if it is currently not displayed!), in order to make sure that its access rights are configured correctly.
- your data security official in charge with your project (e.g., in Germany, the "Datenschutzbeauftragter") gives you, for **every** EntityField, all information you need in order to make sure that possible concerns will be included in the further configurations and programming (e.g., the implementation of a dialog pointing to the "impact on the data privacy information (GDPR)" - see, e.g., method DataPrivacyUtils.notifyNeedDataPrivacyUpdate in DataPrivacy_lib).

8. Creating database tables and columns

Like the Entities, also the database’s structure is based on the data structure we had modeled before (therefore, in principle, the steps of this chapter can be performed independently from the generation of the Entities).

Please remind that, for every database table, an appropriate setting of **database indices** is required, in order to ensure an optimal performance of database access. Find further information about performance optimization in AID066.

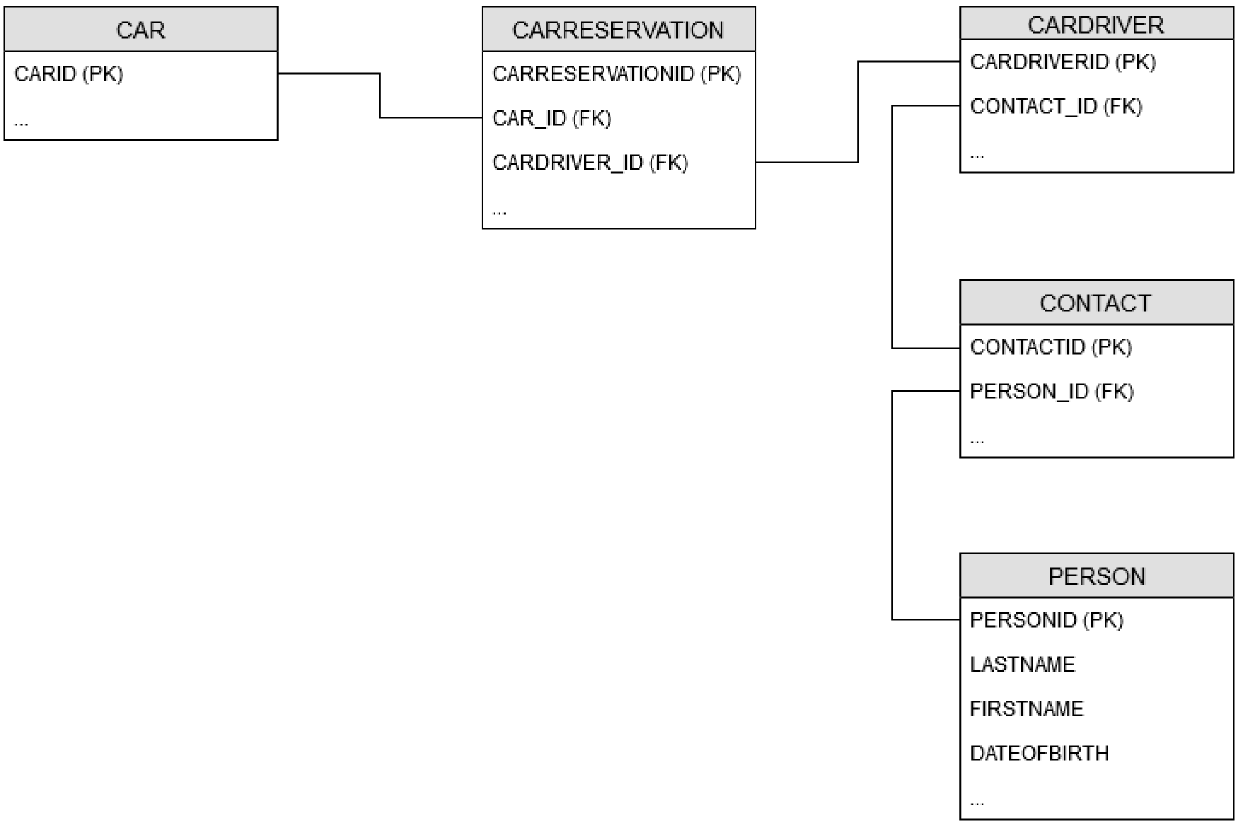


Figure 8. Carpool-related database tables with primary keys (PK) and foreign keys (FK)

Please note: The database tables CONTACT and PERSON already exist, as they are part of the xRM project.

The easiest way to create new tables and columns is to use ADITO’s database editor: In the "Projects" window, navigate to "system" and double-click on the your system’s name, e.g., "default" (ADITO cloud server must be running, and a tunnel connection must be established). This will display your system configuration in an editor window, usually in the upper middle of the Designer. Here, double-click on "Data_alias". Then, your database content will appear as a tree structure, and you can view tables and columns as well as add/edit/delete columns via the context menu when right-clicking on table names or column names.

However, ADITO offers a second way of creating database tables and columns: You define the tables and their columns in an xml file, and then a tool called Liquibase can create the tables and columns automatically, based on the xml file.

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, see https://www.liquibase.org/. Further information can be found in chapter "Create Liquibase files automatically" of the Designer Manual.

8.1. Creating a folder for your xml files

The xml files that Liquibase needs all reside under alias > Data_alias (visible in the "Projects" window). The xml files referring to the xRM project are assigned to a folder called "basic". On the same level as this folder, we now create a new folder reserved for xml files that refer to our project:

Right-click on "Data_alias" and choose New > New folder. Name the new folder "example_carpool".

8.2. Creating an xml file for every table

For every database table we need, create a separate xml file.

- Right-click on the "example_carpool" folder and choose "New" > "New Changeset".
- Changeset name: see below (extension ".xml" will be added automatically)
- Copy the following code into the respective xml file.

create_car.xml

```xml
<?xml version="1.1" encoding="UTF-8" standalone="no"?>
<databaseChangeLog xmlns="http://www.liquibase.org/xml/ns/dbchangelog" xmlns:ext="http://www.liquibase.org/xml/ns/dbchangelog-ext"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog-ext
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-ext.xsd http://www.liquibase.org/xml/ns/dbchangelog
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.6.xsd">
    <changeSet author="j.smith" id="23533445-0d3d-499c-aa98-cf37ca4798c1">
        <createTable tableName="CAR">
            <column name="CARID" type="CHAR(36)">
                <constraints primaryKey="true" primaryKeyName="PK_CAR_CARID"/>
            </column>
            <column name="COLOR" type="VARCHAR(36)"/>
            <column name="LICENSEPLATENUMBER" type="NVARCHAR(20)"/>
            <column name="MANUFACTUREDATE" type="DATE"/>
            <column name="MANUFACTURER" type="VARCHAR(36)"/>
            <column name="PICTURE" type="LONGBLOB"/>
            <column name="PRICE" type="DECIMAL(10,2)"/>
            <column name="CURRENCY" type="VARCHAR(36)"/>
            <column name="TYPE" type="NVARCHAR(30)"/>
        </createTable>
    </changeSet>
</databaseChangeLog>
```

create_cardriver.xml

```xml
<?xml version="1.1" encoding="UTF-8" standalone="no"?>
<databaseChangeLog xmlns="http://www.liquibase.org/xml/ns/dbchangelog" xmlns:ext="http://www.liquibase.org/xml/ns/dbchangelog-ext"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog-ext
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-ext.xsd http://www.liquibase.org/xml/ns/dbchangelog
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.6.xsd">
    <changeSet author="j.smith" id="64fd2d43-8c77-42d4-b349-4ebcd3a45037">
        <createTable tableName="CARDRIVER">
            <column name="CARDRIVERID" type="CHAR(36)">
                <constraints primaryKey="true" primaryKeyName="PK_CARDRIVER_CARDRIVERID"/>
            </column>
            <column name="CONTACT_ID" type="CHAR(36)"/>
            <column name="DRIVINGLICENSENUMBER" type="NVARCHAR(30)"/>
            <column name="DRIVINGLICENSEISSUEDATE" type="DATE"/>
        </createTable>
    </changeSet>
</databaseChangeLog>
```

create_carreservation.xml

```xml
<?xml version="1.1" encoding="UTF-8" standalone="no"?>
<databaseChangeLog xmlns="http://www.liquibase.org/xml/ns/dbchangelog" xmlns:ext="http://www.liquibase.org/xml/ns/dbchangelog-ext"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog-ext
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-ext.xsd http://www.liquibase.org/xml/ns/dbchangelog
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.6.xsd">
    <changeSet author="j.smith" id="45e21347-53cd-48e1-9667-591f3506e9e5">
        <createTable tableName="CARRESERVATION">
            <column name="CARRESERVATIONID" type="CHAR(36)">
                <constraints primaryKey="true" primaryKeyName="PK_CARRESERVATION_CARRESERVATIONID"/>
            </column>
            <column name="CAR_ID" type="CHAR(36)"/>
            <column name="CARDRIVER_ID" type="CHAR(36)"/>
            <column name="STARTDATE" type="DATETIME"/>
            <column name="ENDDATE" type="DATETIME"/>
            <column name="MILEAGERETURN" type="INT"/>
            <column name="PARKINGTICKETFINE" type="DECIMAL(7,2)"/>
            <column name="SPEEDINGFINE" type="DECIMAL(7,2)"/>
            <column name="CURRENCY" type="VARCHAR(36)"/>
            <column name="DAMAGE" type="NVARCHAR(300)"/>
        </createTable>

        <!--Index for speeding up searches / join for CAR_ID, e.g. when searching data by a specific car -->
        <createIndex indexName="IDX_CAR_ID" tableName="CARRESERVATION">
            <column name="CAR_ID"/>
        </createIndex>
        <!--Index for speeding up searches / join for CARDRIVER_ID, e.g. when searching for a spcific driver -->
        <createIndex indexName="IDX_CARDRIVER_ID" tableName="CARRESERVATION">
            <column name="CARDRIVER_ID"/>
        </createIndex>
        <!--Compound index for speeding up joins over CAR_ID and CARDRIVER_ID -->
        <createIndex indexName="IDX_CAR_ID_CARDRIVER_ID" tableName="CARRESERVATION">
            <column name="CAR_ID"/>
            <column name="CARDRIVER_ID"/>
        </createIndex>
    </changeSet>
</databaseChangeLog>
```

Please note that this Liquibase file also includes 3 indices: One for each foreign key (speeding up searches for the respective UID), as well as a compound index, which speeds up SQL JOINs over CAR_ID and CARDRIVER_ID.
The previous Liquibase files (those for tables CAR and CARDRIVER) do not require the explicit configuration of an index, as columns CARID and CARDRIVERID are declared as primary keys, which automatically results in the creation of an index on those columns.
Setting appropriate indices is very important for the system's performance - find

further information in the document AID066 Performance Optimization.

You may change "author" and "id" in tag "changeSet", by inserting your own name and an arbitrary 36-digit UUID, which you can easily generate in the Designer, via option "Tools" > "Generate UUID". (The generated UUID will then be copied to the clipboard, from which you can, as usual, get it via CTRL+V.)

As you can see, we use the data type CHAR(36) in some cases. These columns can be

- the database table’s primary key column, which in ADITO is always named with "ID" as suffix, e.g., "CARID" (mind that there is no underscore before "ID"). This ID column always holds a 36-digit UID.
- a column referencing the primary key of another table. In ADITO, these "foreign key columns" are named like the corresponding primary key column, except for an underscore before the "ID" suffix, e.g. "CAR_ID".

You can find tables with the data types of the database columns corresponding to all car pool related EntityFields in the appendix Car pool example: EntityFields

When Liquibase is executed, Liquibase’s data types (as included in the xml files) are mapped to data types proper to the specific database engine connected to the ADITO project. For example, Liquibase’s data type NCLOB (used for very large text fields) remains a NCLOB for Apache Derby databases, but is mapped to a LONGTEXT for MariaDB and MySQL, while in MicrosoftSQL it will be a NVARCHAR(MAX).
When customizing ADITO, you should always prefer the target data types of these Liquibase mappings, even if you do not use Liquibase itself.
You can find a list of the preferable data types, according to database system, in the article "Preferable data types", available in this article in ADITO Knowledge Base. (To read this article, you need access to the ADITO Service Client.)

In the database of the ADITO xRM project, constraints are usually only set for the primary key, and in very few further cases (e.g., a "not null" constraints for all columns refering to an EntityField of contentType "Boolean"). In particular, there are no foreign key constraints on database level. If you want to make sure that a specific EntityField is not empty, you usually set its property "mandatory" to true (rather than setting a "not null" constraint on its corresponding database column).
This has multiple reasons, e.g., it simplifies the task of dropping and re-creating database tables, and it grants more flexibility when creating interdependent datasets.

8.3. Including xml files in changelog

Now we must include the new create_xxx.xml files in ADITO’s automatic database management, which is based on Liquibase. This tool executes all tasks defined in the file changelog.xml, which resides in the folder alias > Data_alias (see "Projects" window). This "main" changelog.xml file, in turn, references

- further files with the same name **changelog.xml**, on lower folder levels (mostly containing the definition of tables and their columns)
- files named **init.xml** or **init_xxx.xml** (mostly containing data to be inserted into tables, e.g., configuration data or example data)

In the case of our new create_xxx.xml required for the car pool management, we first create a new changelog.xml file, and then reference it in the main changelog.xml file:

In folder example_carpool (i.e., in parallel to the new create_xxx.xml files), create an empty xml file named changelog.xml (file type: "changelog.xml") and insert the following code:

changelog.xml (under alias > Data_alias > example_carpool)

```
<?xml version="1.1" encoding="UTF-8" standalone="no"?>
<databaseChangeLog xmlns="http://www.liquibase.org/xml/ns/dbchangelog" xmlns:ext="http://www.liquibase.org/xml/ns/dbchangelog-ext"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog-ext
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-ext.xsd http://www.liquibase.org/xml/ns/dbchangelog
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.6.xsd">
    <include relativeToChangelogFile="true" file="create_car.xml"/>
    <include relativeToChangelogFile="true" file="create_cardriver.xml"/>
    <include relativeToChangelogFile="true" file="create_carreservation.xml"/>
</databaseChangeLog>
```

This new changelog.xml file must now be referenced in the main changelog.xml (under alias > Data_alias) by adding the following additional code line:

changelog.xml (under alias > Data_alias)

```
...
<include relativeToChangelogFile="true" file="example_carpool/changelog.xml"/>
...
```

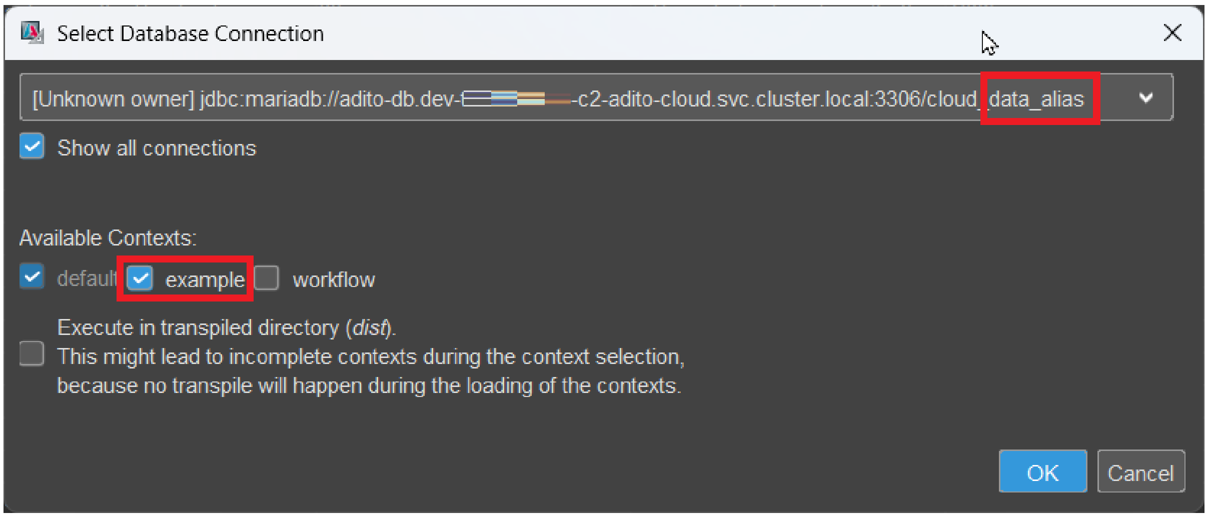
Make sure you do not confuse the multiple changelog.xml files due to their equal name.

8.4. Liquibase update

Now you can execute a command that will update the database structure on the basis of the xml files we have just added:

alias > Data_alias > (context menu:) Liquibase > Update…
(This option will only be available, if a database connection exists.)

This will open a dialog, in which you select your database connection (it is named "...cloud_data_alias"), check option "example" (it can take a while until it appears) and confirm by "OK".



Select Database Connection

[Unknown owner] jdbc:mariadb://adito-db.dev-[redacted]-c2-adito-cloud.svc.cluster.local:3306/cloud_data_alias

[✓] Show all connections

Available Contexts:

[✓] default [✓] example [ ] workflow

[ ] Execute in transpiled directory (*dist*).
This might lead to incomplete contexts during the context selection,
because no transpile will happen during the loading of the contexts.

As you can see in the above screenshot, you can decide, whether or not example data (contacts, companies, Activities, etc.) should be inserted in your application. Please be aware: If you have not inserted this example data earlier, checking checkbox "example" will now result in a **complete loss of any productive data - even if you only choose option "Liquibase - update"** (without "drop all"). Therefore, checkbox "example" should NEVER be checked in a productive system or whenever you have entered your own data that must not be deleted. (However, for working with the carpool project, "example" should be checked, because, e.g., demo data of Context "Contact" (PERSON) are required for being referenced in Context "CarDriver".)

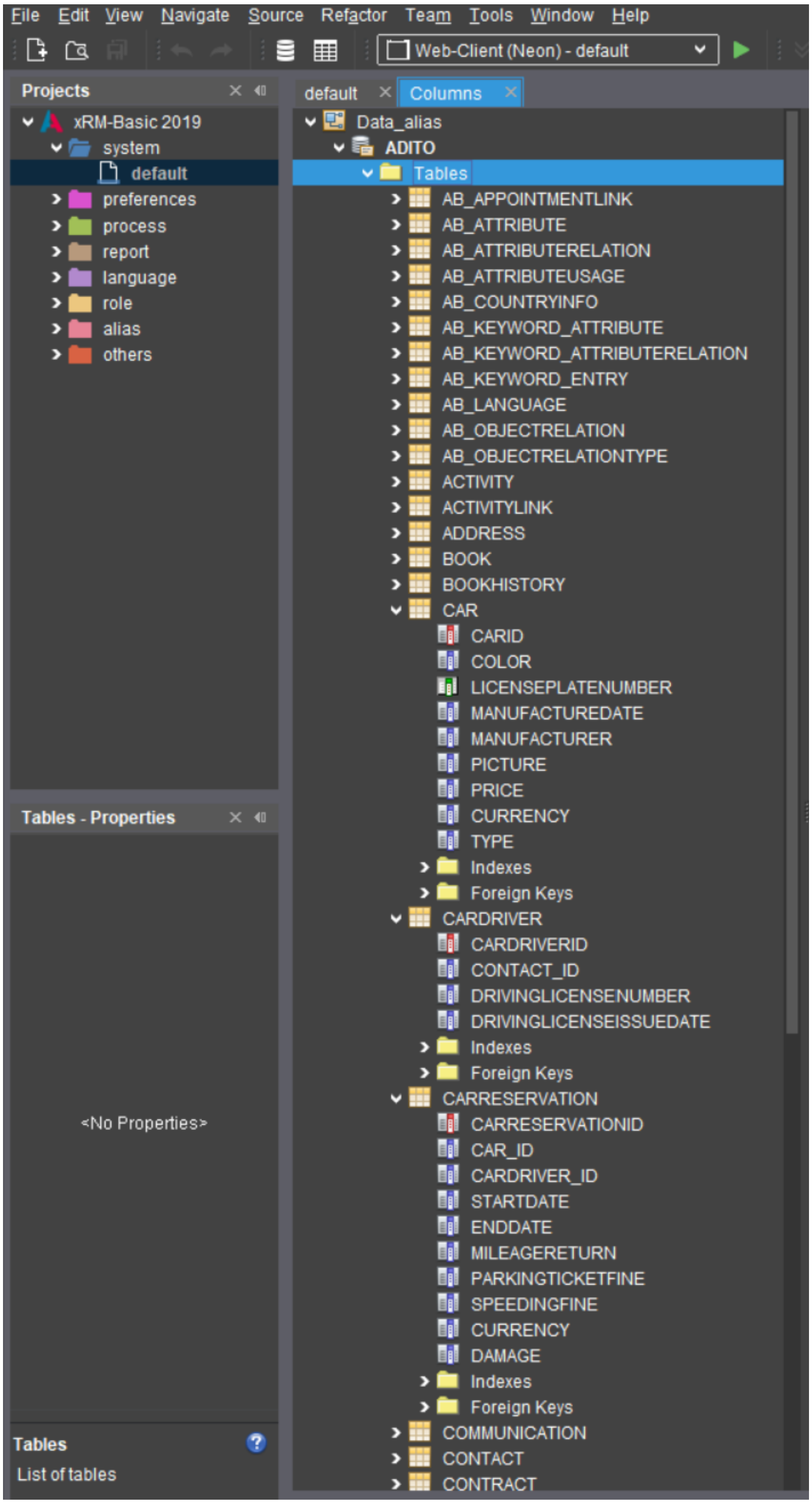
If everything has been configured correctly, you will, after a few seconds, read "Update successful!" in a small message window (called "Balloon") in the lower right corner of the Designer. In rare cases, you might get an error message, if you select "Update...". If so, choose "Drop All & Update..." instead. This will - in addition to the creation of the CAR-related tables - delete (drop) *all* tables of the xRM project and build them again, including the example data.

If one of the liquibase xml files (also called "changesets") contains an error (e.g., a typo), the update process stops at this file, and the following liquibase files are not being executed. There is no rollback in this case. If you choose "Drop All & Update...", then "Drop All" and "Update" are separate commands, which are executed subsequently. If, e.g., "Drop All" has been executed, but the first "Update" xml file fails, then the database is empty. A single changeset is always executed as database transaction, i.e., if, e.g., in a table creation file, the third column has been

> misconfigured, then the whole table is not created.

You can now check, if all tables have been created correctly, using ADITO’s database editor: Make sure that there is a database connection. In the "Projects" window, double-click on system > default. As next step, you sometimes have to click the "Connect" or "Reconnect" button in the editor window (upper middle part of the Designer). Double-click on "Data_alias". Then you will see the database structure in a tree. Navigate to Data_alias > adito_data > Tables to view all tables of your ADITO project (except the system tables, see below).

The car pool related database tables and columns should appear as follows:


File Edit View Navigate Source Refactor Team Tools Window Help

Web-Client (Neon) - default

Projects

- xRM-Basic 2019
  - system
    - default
  - preferences
  - process
  - report
  - language
  - role
  - alias
  - others

Tables - Properties

<No Properties>

Tables

List of tables

default | Columns

- Data_alias
  - ADITO
    - Tables
      - AB_APPOINTMENTLINK
      - AB_ATTRIBUTE
      - AB_ATTRIBUTERELATION
      - AB_ATTRIBUTEUSAGE
      - AB_COUNTRYINFO
      - AB_KEYWORD_ATTRIBUTE
      - AB_KEYWORD_ATTRIBUTERELATION
      - AB_KEYWORD_ENTRY
      - AB_LANGUAGE
      - AB_OBJECTRELATION
      - AB_OBJECTRELATIONTYPE
      - ACTIVITY
      - ACTIVITYLINK
      - ADDRESS
      - BOOK
      - BOOKHISTORY
      - CAR
        - CARID
        - COLOR
        - LICENSEPLATENUMBER
        - MANUFACTUREDATE
        - MANUFACTURER
        - PICTURE
        - PRICE
        - CURRENCY
        - TYPE
        - Indexes
        - Foreign Keys
      - CARDRIVER
        - CARDRIVERID
        - CONTACT_ID
        - DRIVINGLICENSENUMBER
        - DRIVINGLICENSEISSUEDATE
        - Indexes
        - Foreign Keys
      - CARRESERVATION
        - CARRESERVATIONID
        - CAR_ID
        - CARDRIVER_ID
        - STARTDATE
        - ENDDATE
        - MILEAGERETURN
        - PARKINGTICKETFINE
        - SPEEDINGFINE
        - CURRENCY
        - DAMAGE
        - Indexes
        - Foreign Keys
      - COMMUNICATION
      - CONTACT
      - CONTRACT

Figure 9. The carpool-related database tables and their columns

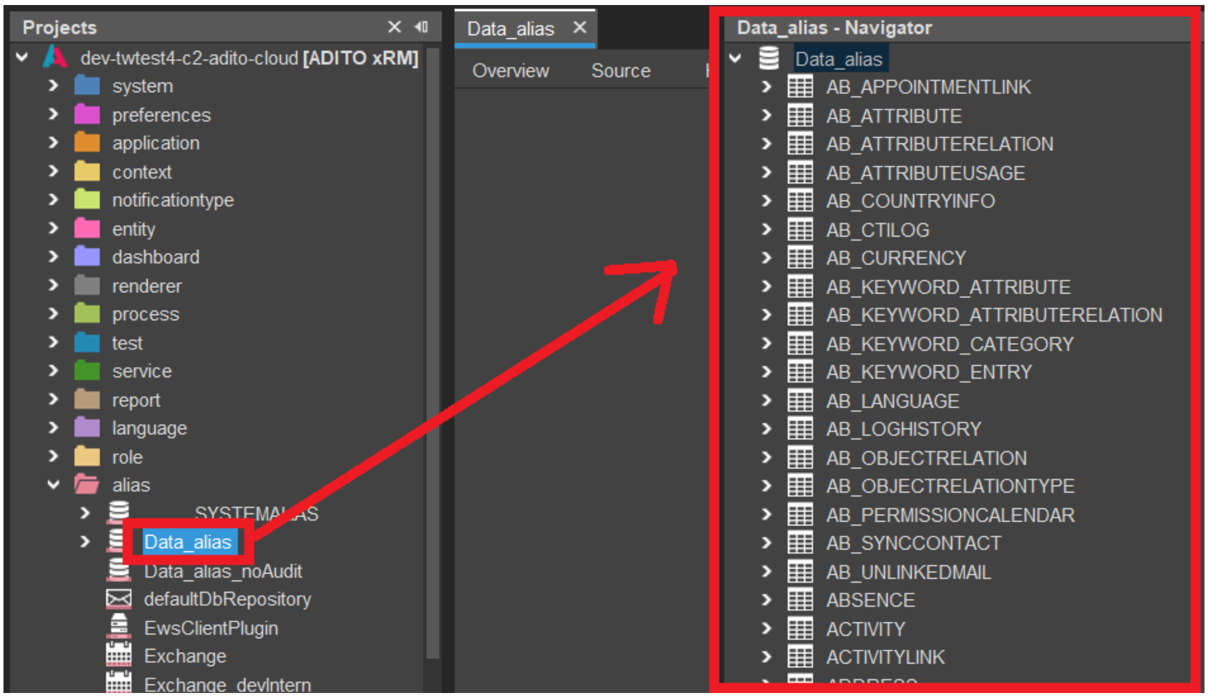
Make sure that you named database tables and columns exactly as shown above. Otherwise, some code snippets of this manual might not work.

Furthermore, the demo data of the ADITO xRM project has been inserted by the Liquibase update (e.g., data of persons and companies).

8.5. Updating the Alias Definition

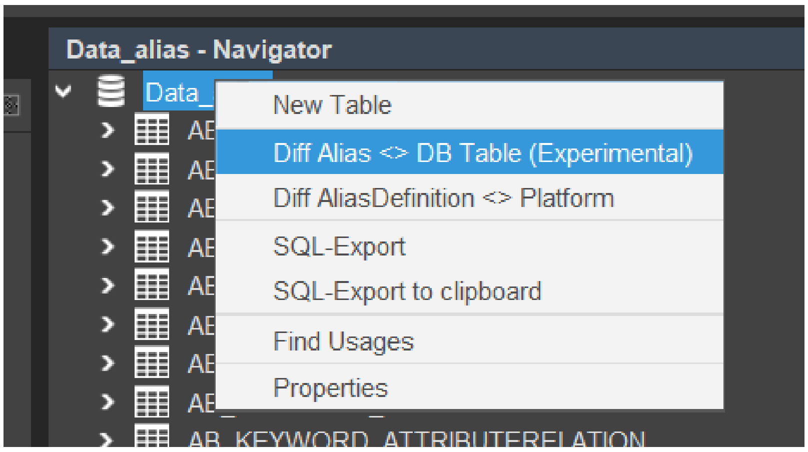
The Liquibase update we have just executed only affected the database itself. In order to adapt an ADITO project to these database changes, we must execute a second update:

In the "Projects" window, under "alias", double-click on "Data_alias". Now, in the "Navigator" window, you see the so-called "Alias Definition", which, so far, does not include the new tables:



The "Alias Definition", also called "Database structure of the project" is, in principle, a kind of copy of the database structure, augmented with additional features, i.e., you can optionally assign additional properties to a table or to a column (properties, which cannot be assigned in the database itself, e.g. a title or a description).

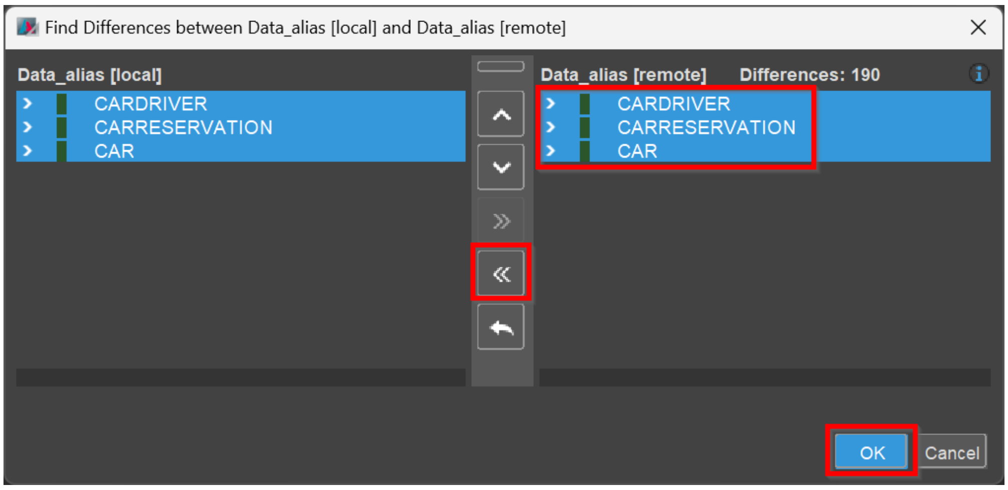
In the "Navigator" (!) window, right-click on "Data_alias" and select "Diff Alias <> DB Table":


Data_alias - Navigator
Data_
AB_KEYWORD_ATTRIBUTERELATION
New Table
Diff Alias <> DB Table (Experimental)
Diff AliasDefinition <> Platform
SQL-Export
SQL-Export to clipboard
Find Usages
Properties

In the following dialog, select your project and your database alias (e.g., "default"), and then "OK". A larger dialog will open, showing all differences between the database structure ("Data_alias [remote]", right part) and the Alias Definition ("Data_alias [local]", left part).

If you hover with the mouse pointer over the little "i" icon in the upper right corner of this dialog, a legend will pop up, explaining the meaning of the colors of the little bars shown in front of each table name.

As our Liquibase update has made the database’s structure "newer" than the Alias Definition, we now need to perform an update from "remote" (right part) to "local" (left part): Mark all 3 carpool-related table names shown on the right (click on them subsequently, holding the CTRL key). Click on the button showing 2 arrows pointing to the left (<<), followed by "OK".



The update process might take some time. After it has finished, look at the "Navigator" window: The new tables and their columns have been added to the Alias Definition.

If you want to create or modify your database structure without using Liquibase, you can always choose between both ways:

- **Either** you start with the database editor (system > default > Data_alias > ADITO), create your tables and columns (or you create it directly via SQL - see "Execute SQL" function via the button to the left of the combo box in the button bar), and then update the Alias Definition via "Diff Alias <> DB Table" from "remote" to "local".
- **Or** you do it the other way round: You first create your tables and columns in the Alias Definition: Double-click on alias > Data_alias and then choose "Create Table" from the context menu when right-clicking on "Data_alias" in the Navigator window. Then right-click on the new table and choose "New Column". Afterwards, click on the new column and edit its properties according to your specification. Finally, update the database via "Diff Alias <> DB Table" from "local" to "remote".

In both cases, it might be useful to auto-generate Liquibase files afterwards, because then, e.g., you can reset your database via the Liquibase-related function "Drop all & Update…" (see above). Please refer to chapter "Create Liquibase files automatically" of the Designer Manual.

8.6. Connecting EntityFields with database columns (RecordContainer)

Now that both the Entities and the database structure exist, we have got to "tell" ADITO, how specific EntityFields are corresponding with specific database columns, acting as data source.

For general information about how to specify what data is to be loaded, please refer to appendix Database Access, chapter "Basic SQL Statement".

The "connection" between EntityFields and database columns is configured in a so-called RecordContainer, which must once be created for every Entity:

Double-click on an Entity (under "entity" in the "Projects" window), e.g., on Car_entity. Then the Entity appears in the Navigator window. Here, right-click on the Entity’s name and choose option "New RecordContainer" from the context menu. In the following dialog, select "dbRecordContainer" as type and enter any name, e.g. "db" (for database), followed by "OK". This creates a RecordContainer for connecting EntityFields with database columns as data source. (As you could see in the "Type" combo box, there are also RecordContainers having jDito code as data source. Their usage will be explained later.)

This new RecordContainer’s name is automatically inserted in the Entities' property "recordContainer".

At first, we handle those EntityFields which simply show the value of a specific database column. These fields can easily be recognized, as we had written their names in uppercase letters, and the name is the same as the name of the corresponding database column. To "connect" these kinds of EntityFields with a database column, we initially specify the respective database table: Click on the RecordContainer’s name "db" (under Car_entity > RecordContainers in the Navigator window) and set the property "alias" to "Data_alias". Now, ADITO knows which database to access.

Then edit the property "linkInformation" by clicking on the three-dotted button to the right of the property line: A dialog appears, in which you click on the plus sign ("+") and select "CAR" in column "Table". The next column "Primary key" is filled automatically. Leave the checkbox in column "UID Table" checked. (This will automatically insert a new UID in the primary key field, whenever a new dataset is created - besides some other effects, which will be explained later.) Leave the checkbox in column "Read only" *un*checked. Confirm with "OK".

Now, ADITO knows that the data of Car_entity is to be accessed via the database table CAR and its columns. In SQL terms: "CAR" is to be used in the "from" clause.

If you open the node "db", you see two nodes for every EntityField: One named "…value" (to be saved in the database, and to be used for calculation purposes), one named "…displayValue" (to be used exclusively for displaying purposes). These nodes are called "RecordFieldMappings", sometimes abbreviated as only "RecordFields". For now, we only need the "values", not the "displayValues". (If the displayValues are not set, ADITO uses the values also as displayValues.)

In order to configure a RecordFieldMapping, you first need to initialize it, by double-clicking on it or by right-clicking on it and choosing option "Initialize" from the context menu. Its font color will change from grey to white. (If you want to undo the initialization, right-click on the RecordFieldMapping again and then choose option "Restore Default Value". This will reset its font color from white to grey again, indicating that it is not initialized.)
Please make sure that you initialize only those RecordFieldMappings whose corresponding EntityFields need to be connected to the database. If you initialize RecordFieldMappings unnecessarily (e.g., for calculated fields), the performance of ADITO will be decreased.

Click on RecordFieldMapping, e.g., CARID.value, and set the corresponding database column in property "recordField" (CAR.CARID, CAR.COLOR, etc.). Repeat this step for all other "value" RecordFieldMappings showing EntityField names in uppercase letters. Now, ADITO knows what database columns to access. In SQL terms: "CARID", "COLOR", etc. are to be used in the "select" clause.

Furthermore, check property "isFilterable", if you want the EntityField to be available as filter criteria in the filter component of the FilterView (remember to repeat this, in case you later set a displayValue). Then, ADITO automatically adds the corresponding "where" clause to the SQL statement.

Repeat the previous steps also for the other tables.

> The above property configuration is the basic way in ADITO to establish an automated database access. This means, depending on the requirement, ADITO automatically creates the suitable SQL statement, be it SELECT, INSERT, UPDATE, or DELETE including the required "from" clause and (if a filter has been set) the "where" clause. Optionally, this basic automatism can be modified by an advanced configuration, using various additional properties, e.g., for setting an arbitrary "where" clause (property "whereClauseProcess" of the dbRecordContainer). Find further information in chapter Database RecordContainer and in the appendix Database Access.

EntityFields not simply showing the value of a database column (e.g., because their value is calculated from other fields) will be handled later, in chapter [Calculated Fields].

8.7. Using database views

(Excursus)

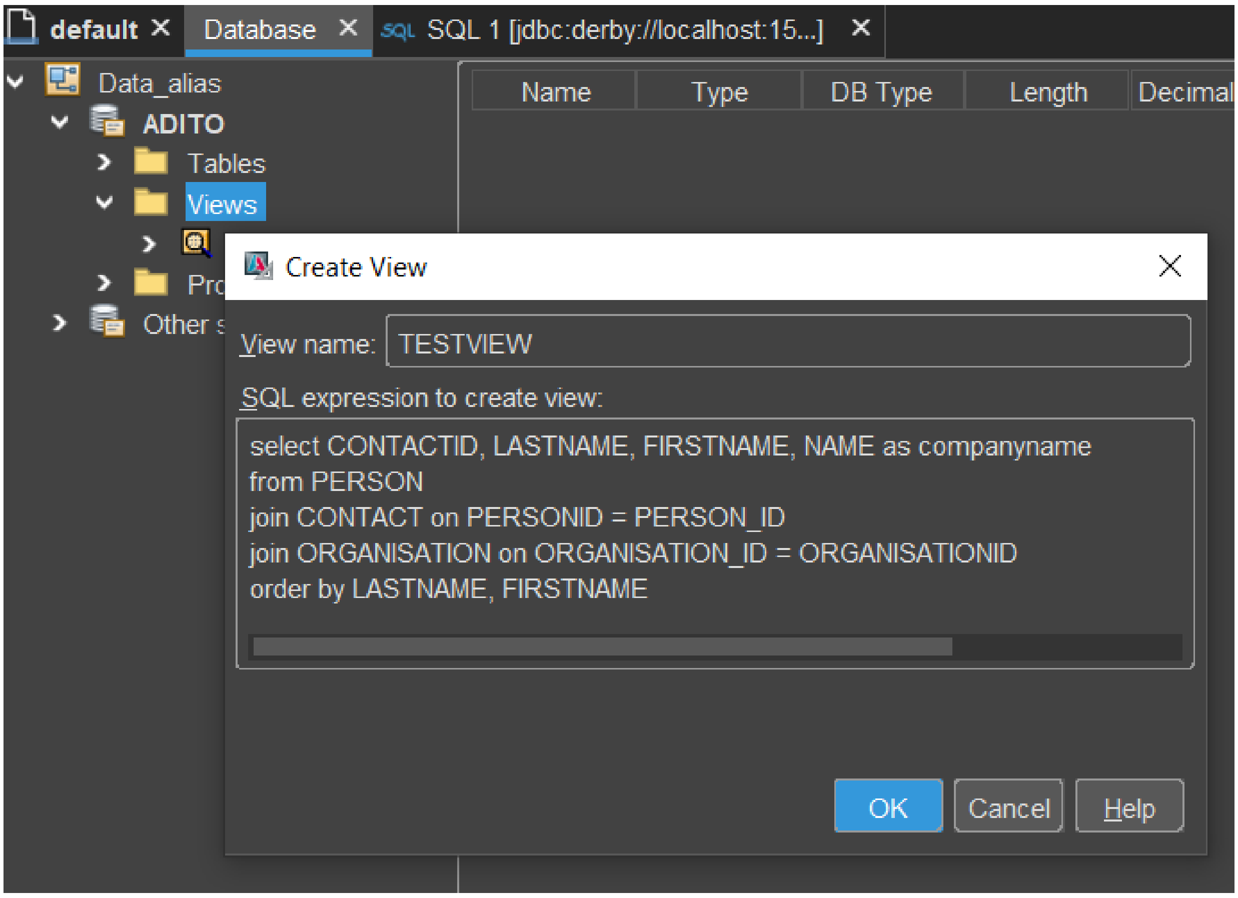
> This chapter is about database views, not about the ADITO model named "View".

Besides database tables, an Entity can also be related to a database view. To explain how to establish this, we will use a simple example of testing purposes: We want to connect TestEntity_entity to a database view named TESTVIEW, which shows every PERSON dataset along with the related company name (ORGANISATION.NAME).

To realize this, proceed as follows:

- Create the database view, e.g., in the Designer’s in-built database editor:
  - Navigate to system > default > Data_alias. In the database tree, open node ADITO, right-click on sub-node "Views" and choose "Create View…" from the context menu.
  - Enter the view’s name TESTVIEW and the SQL select to create it

```
select CONTACTID, LASTNAME, FIRSTNAME, NAME as companyname
from PERSON
join CONTACT on PERSONID = PERSON_ID
join ORGANISATION on ORGANISATION_ID = ORGANISATIONID
order by LASTNAME, FIRSTNAME
```

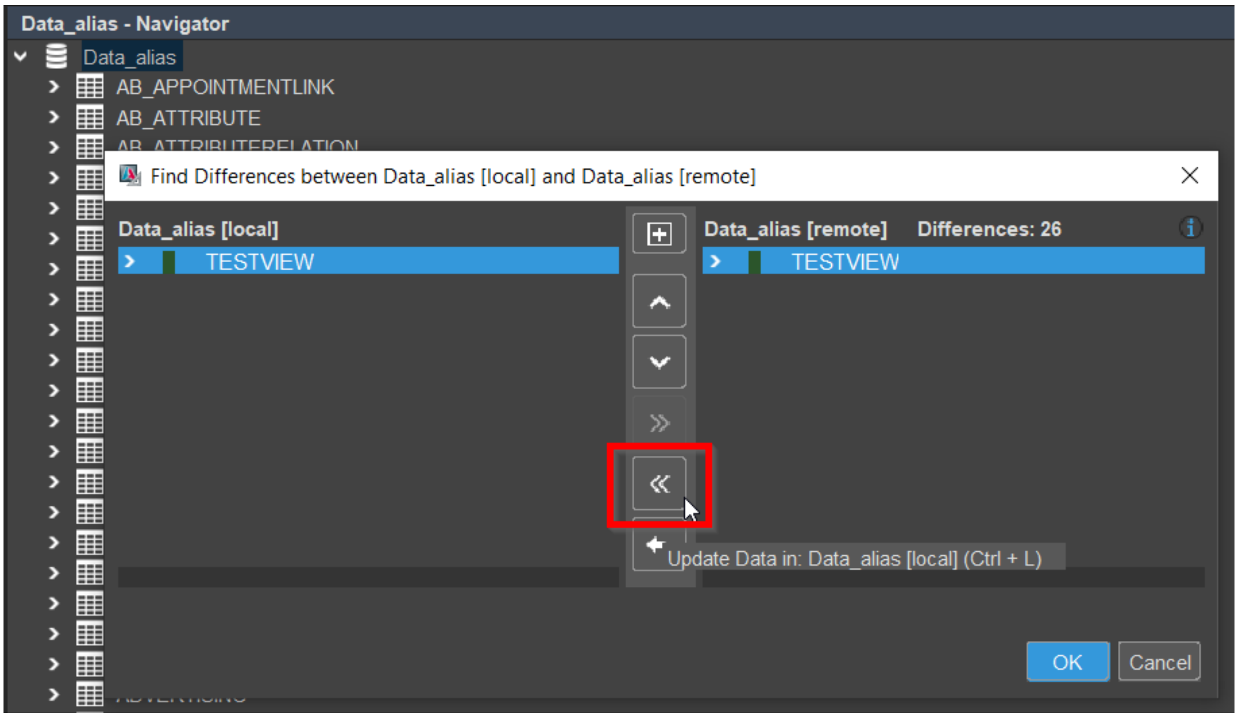


View name: TESTVIEW

SQL expression to create view:

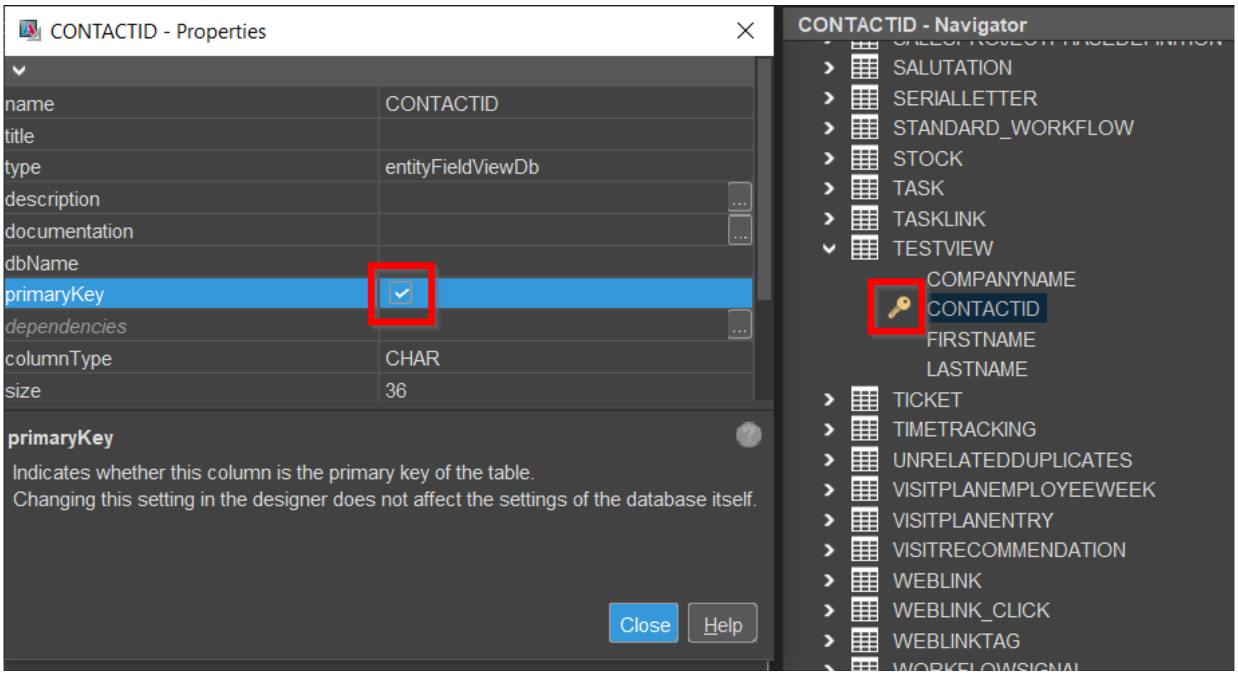
```
select CONTACTID, LASTNAME, FIRSTNAME, NAME as companyname
from PERSON
join CONTACT on PERSONID = PERSON_ID
join ORGANISATION on ORGANISATION_ID = ORGANISATIONID
order by LASTNAME, FIRSTNAME
```

- Update the AliasDefinition via option "Diff Alias <> DB Table", as explained in the previous chapter.



In the case of database views, an update is only possible in this direction. The other direction - creating a view in the AliasDefinition and then updating it into the database - is not possible (this is only possibly for "real" database tables, as explained in the previous chapter).

- In the AliasDefinition, mark the primary key column of the database view, by setting its property "primaryKey" to true.



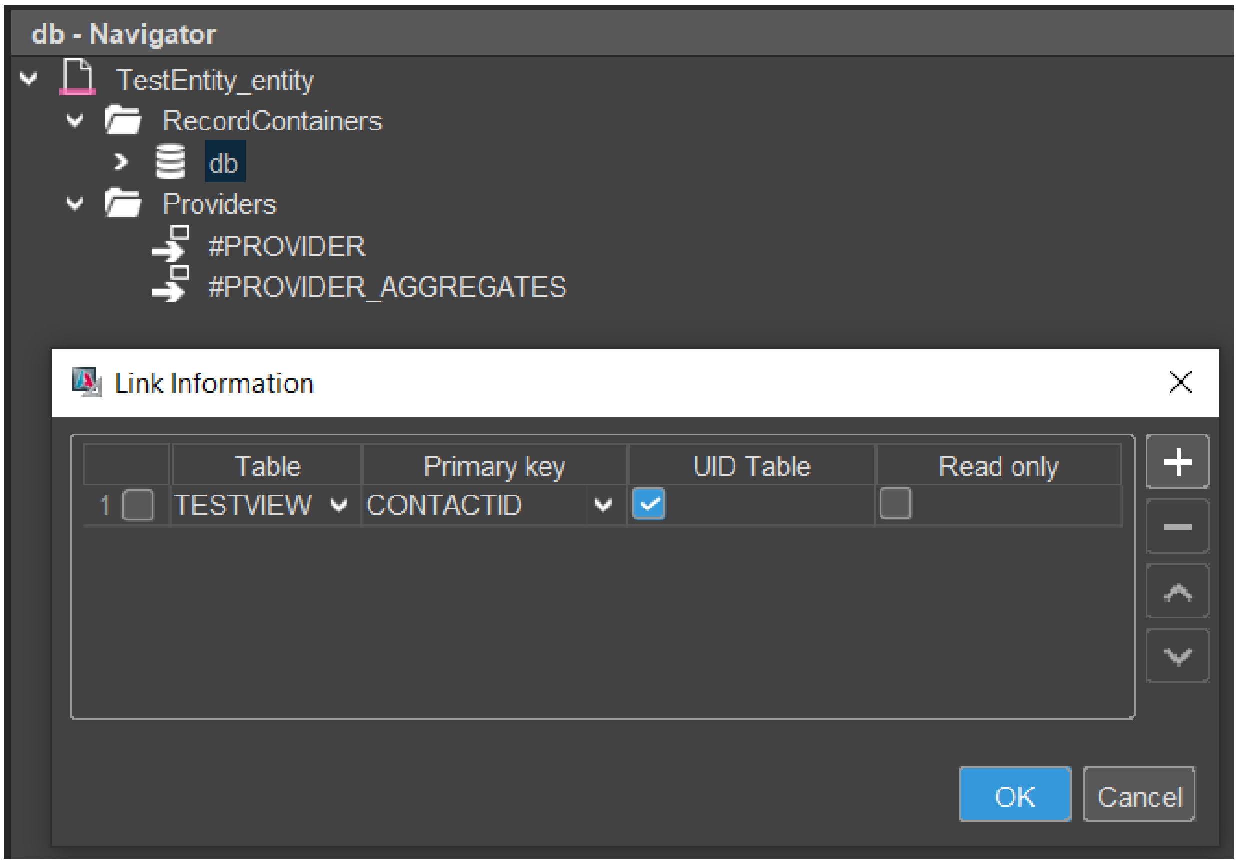
name	CONTACTID
title	
type	entityFieldViewDb
description	
documentation	
dbName	
primaryKey	✓
dependencies	
columnType	CHAR
size	36

primaryKey

Indicates whether this column is the primary key of the table.
Changing this setting in the designer does not affect the settings of the database itself.

SALUTATION
SERIALLETTER
STANDARD_WORKFLOW
STOCK
TASK
TASKLINK
TESTVIEW
COMPANYNAME
CONTACTID
FIRSTNAME
LASTNAME
TICKET
TIMETRACKING
UNRELATEDDUPLICATES
VISITPLANEMPLOYEEWEEK
VISITPLANENTRY
VISITRECOMMENDATION
WEBLINK
WEBLINK_CLICK
WEBLINKTAG

- Create TestEntity_entity with EntityFields according to the database view’s columns and with a DbRecordContainer. After setting the RecordContainer’s property "alias" to "Data_alias", you will be able to select the new database view in property "linkInformation". If you do not find the view there, you have either forgot to update the AliasDefinition or to set its primary key (see previous steps).

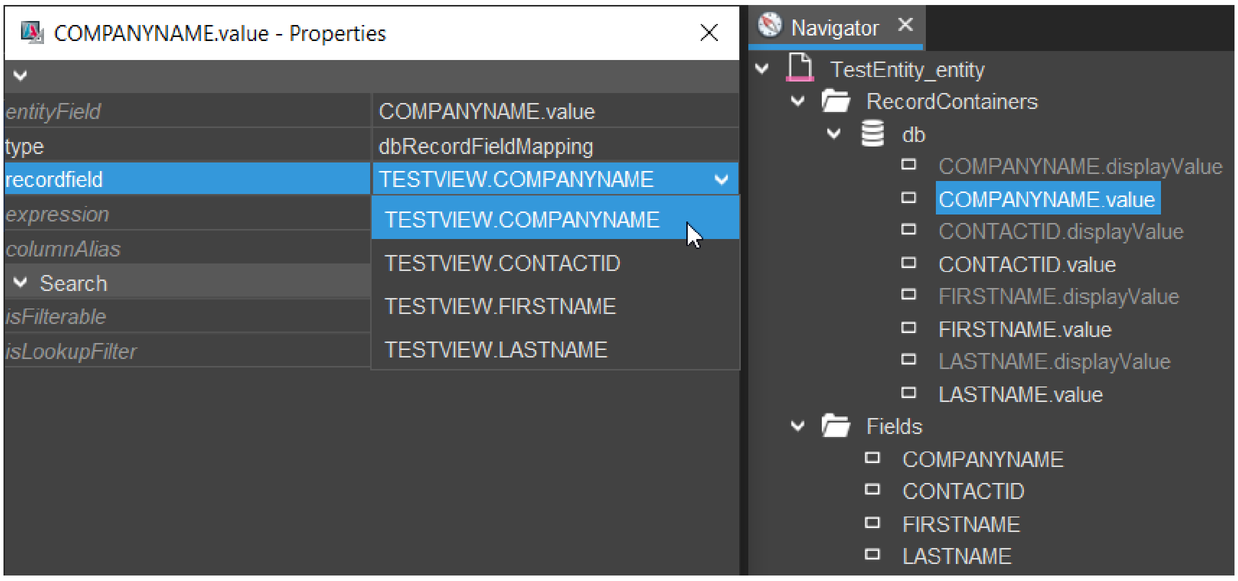

db - Navigator

- TestEntity_entity
  - RecordContainers
    - db
  - Providers
    - #PROVIDER
    - #PROVIDER_AGGREGATES

Link Information

	Table	Primary key	UID Table	Read only
1 ☐	TESTVIEW	CONTACTID	✓	☐

- Now you can open the sub-nodes of the RecordContainer (RecordFieldMappings) and assign the EntityFields to the corresponding columns of the database view.


COMPANYNAME.value - Properties

entityField	COMPANYNAME.value
type	dbRecordFieldMapping
recordfield	TESTVIEW.COMPANYNAME
expression	
columnAlias	
Search	
isFilterable	
isLookupFilter	

TESTVIEW.COMPANYNAME
TESTVIEW.CONTACTID
TESTVIEW.FIRSTNAME
TESTVIEW.LASTNAME

Navigator

- TestEntity_entity
  - RecordContainers
    - db
      - COMPANYNAME.displayValue
      - COMPANYNAME.value
      - CONTACTID.displayValue
      - CONTACTID.value
      - FIRSTNAME.displayValue
      - FIRSTNAME.value
      - LASTNAME.displayValue
      - LASTNAME.value
  - Fields
    - COMPANYNAME
    - CONTACTID
    - FIRSTNAME
    - LASTNAME

9. Making data visible

The preceding chapters have covered exclusively the data structure and its representation in the project. The previously created elements (database tables and their columns, Entities and their EntityFields, RecordContainers and their RecordFieldMappings) can all be considered as part of a "backend", i.e., they alone do not yet appear in the browser ("frontend"). To make them and their data visible, we use so-called Views (not to be confused with database views), which in turn are clustered in so-called Contexts. Every Context is related to one Entity and usually contains different Views for different purposes, e.g., for displaying, filtering, and editing data.

As no View can exist alone, but needs a Context to be assigned to, we first create Contexts for every Entity, and afterwards we create Views "inside" Contexts.

9.1. Creating Contexts

In the "Projects" window, right-click on "context" and select option "New" in the context menu. This will open a dialog, in which you select your project, enter a suitable name (Context names are written in CamelCase, starting with an uppercase letter - here, we write Car, CarDriver, and CarReservation), and select "Context" as type. Confirm with "OK". Perform this step for creating the Contexts Car, CarDriver, and CarReservation. Then, connect the new Contexts with the corresponding Entities, by setting each Context’s property "entity" accordingly (e.g., select "Car_entity" for Context Car, etc.) . This assignment will immediately be shown by a sub-node showing the Entity name, under the Context name.

The normal workflow for creating ADITO applications is that you first create a new Entity and then a Context. However, in some cases it may be helpful to know that you can do it also the other way round: You can first create a Context and then, directly "in" the Context, a new Entity (right-click on the Context, select "New" from the context menu, and then - in the "Create New Model" dialog - select "entity" in combo box "Type"); the Context’s property "entity" will then be set automatically.

In an ADITO project, every Entity belongs to exactly one Context. Do never assign one Entity to more than one Context (although, in earlier ADITO versions, this was technically possible), as this will confuse the logic and cause exceptions.

9.2. Views

Each Context usually contains at least the following basic Views:

- "FilterView", for displaying an Entity’s data in a table structure, along with the option to filter the data according to specific criteria

- "PreviewView", for showing details of the dataset marked in the FilterView
- "MainView", for displaying one single set of an Entity's data, often along with reference data from other Contexts
- "EditView", for editing the EntityFields of one single dataset

Depending on the purpose of the Context, there may be more or less Views.

9.2.1. Creating Views

To create a new View, right-click on a Context and choose option "New" in the context menu. This will open a dialog, in which you select your project, a suitable name, and "view" as type. According to the conventions given in ADITO's spelling guidelines (see ADITO Information Document AID001, chapter "Spelling & Wording" > "ADITO models"), the name of the View starts with the name of the Context, followed by its purpose in CamelCase, and ending with the suffix "_view".

Now, create the above mentioned basic Views for each Context, naming them according to the convention. Thus, e.g., in case of the Context Car, the names are "CarFilter_view", "CarPreview_view", "CarMain_view", and "CarEdit_view".

Once all Views are created, they must be assigned to the Contexts' properties "mainview", "filterview", "editview", and "preview".

9.2.2. Assigning layout and ViewTemplates

The appearance of a View cannot be designed arbitrarily. Rather, its visible structure is determined by

- a predefined layout, selectable in the combo box of the View's property "layout";
- one or more predefined ViewTemplates.

> This chapter gives only a rough introduction to the topic, along with a few examples. Find detailed information on layouts and ViewTemplates in the sub-chapters of chapter Controlling the design.

First, a layout is set, by selecting from the combo box in the View's property "layout". Some Views require that additional properties are set. E.g., a FilterView needs its property "filterable" to be set to true. See table below.

To assign a ViewTemplate, double-click on the View in the "Projects" window and then right-click on the View in the Navigator window. Select option "Add ViewTemplate..." in the context menu. A dialog will open, in which you select a suitable template type on the left (see table below), leave the field "Assign to" empty, enter a name for the ViewTemplate, and confirm with "OK". After you have assigned the template, it appears as sub-node under the View (in the Navigator window). You can click on it to see its properties.

Layouts and ViewTemplates can be combined in various ways. However, in the first approach, the following standard configuration of ViewTemplates, layouts, and further properties is suitable in many cases, so we will use it for our car pool example:

Table 1. Example configuration for Views and ViewTemplates of Car_entity

View	View properties	Template type	Template properties
FilterView	Layout: GroupLayout filterable: true	Table, TreeTable	entityField: #ENTITY columns: e.g., CARID, MANUFACTURER, TYPE, COLOR
PreviewView	Layout: HeaderFooterLayout header: Card footer: Generic *(values can be set as soon as these ViewTemplates are generated)*	Card (assign to header) Generic ScoreCard (assign to footer)	entityField: #ENTITY iconField: PICTURE titleField: e.g., MANUFACTURER subtitleField: e.g., TYPE descriptionField: e.g., COLOR informationField: e.g., mileage entityField: #ENTITY fields: e.g., MANUFACTUREDATE, PRICE, etc. showDrawer: true drawerCaption: Details hideEmptyFields: false (checkbox unchecked) entityField: #ENTITY fields: e.g., LICENSEPLATENUMBER
EditView	Layout: BoxLayout	Generic	entityField: #ENTITY editMode: true fields: all fields in uppercase letters (or part of them), except the primary key (e.g., CARID)

View	View properties	Template type	Template properties
MainView	Layout: MasterDetailLayout master: PreviewView (see below)	not required (reference to PreviewView)	

Setting a ViewTemplate’s property "entityField" to "#ENTITY" means that all fields of the Entity can be loaded and are therefore available in all EntityField-related properties, e.g., "columns" or "fields". If you actually need only one single EntityField, you should select it in property "entityField" accordingly, because this will restrict the loading process and therefore result in a better performance.

As you can see that it is common to assign at least 2 different ViewTemplates to a PreviewView, so it appears with a "header" area ("Card", showing the main EntityFields and often a picture) and an "footer" area ("Generic", showing further EntityFields). That is why this View layout is called "HeaderFooterLayout". All further ViewTemplates that you might add to the View would be displayed in the middle, i.e., between "header" and "footer".

Additional information on ViewTemplate "Generic":

- Property hideEmptyFields controls whether or not a line with the label (title) of an EntityField is still to be displayed, even if the EntityField has no value (= if it is "empty"). For PreviewViews, we choose to set this property to false, as this ensures a uniform layout that is independent from the dataset marked in the FilterView. But, of course, the setting of this property is ultimately up to your customer’s requirements.
- Property isLabelPositionTob controls if the label is to be shown above the value (true) or to the left of it (false). Default value is false, which fits in most cases.

In many cases, the footer of a PreviewView is a ScoreCardViewTemplate, see chapter ScoreCard. In our car pool example, we will configure this later, for the EntityField "availability" (see chapter Example: Availability).

Furthermore, you see that a MainView usually has no own View, but it references other Views. Its layout is usually a "MasterDetailLayout", with the Context’s PreviewView acting as "Master" and Views of other Contexts acting as "Details" (in order to display data dependent from the data shown in the "Master"). To add a reference to other Views, right-click on the View in the Navigator window, and select "Add reference to existing View…". A dialog will open, in which you select "#ENTITY" as EntityField (this spelling means, that **all** Views of the current Entity will be selectable in the next line), the PreviewView as View, and "master" in line "Assign to". This will automatically set the MainView’s property "master" to the respective PreviewView. How to add the "Detail" part of the "MasterDetailLayout" will be explained in a later chapter ("Complex dependencies").

As for the ViewTemplate "Table", there is a property named "linkedColumns": Here you can set an arbitrary number of columns that are to be provided with a hyperlink to the MainView. The hyperlink functionality will be displayed by a blue font color. If "linkedColumns" is not set, the hyperlink will automatically be assigned to the first column set in property "columns".
Furthermore, the ViewTemplate "Table" has a property named hideContentSearch. If you set this property to false, a lookup bar will appear on the top of the table. This bar is named content search (Context filter). If you start typing in this bar, the datasets are filtered accordingly. This only works, if you have checked property "isLookupFilter" of every RecordFieldMapping whose related EntityField should be included in the filtering. (It does not work EntityFields of all content types, e.g., it does not work for date values, but it works for text values.)

Now, repeat the above steps in order to create and configure the Views of the Entities CarDriver_entity and CarReservation_entity. In the ViewTemplates, you may set arbitrary fields/columns, but, in this first approach, we recommend to use only those EntityFields that written in uppercase letters.

9.2.3. Blueprints

(Excursus)

"Blueprint" is a functionality to simplify the creation of ADITO models, e.g., Entities, EntityFields, Contexts, and Views. You can execute Blueprints in the "Projects" window, via the context menu of the nodes "context" or "entity", respectively:

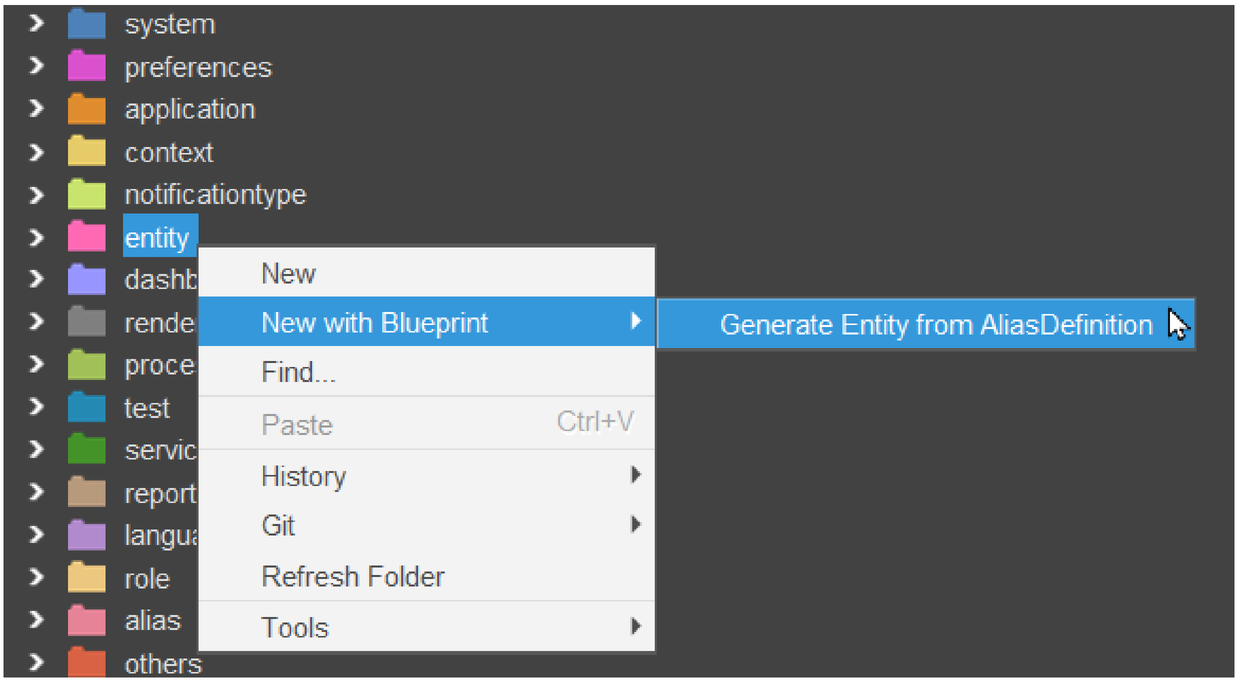


Figure 10. Blueprint available for node "entity"

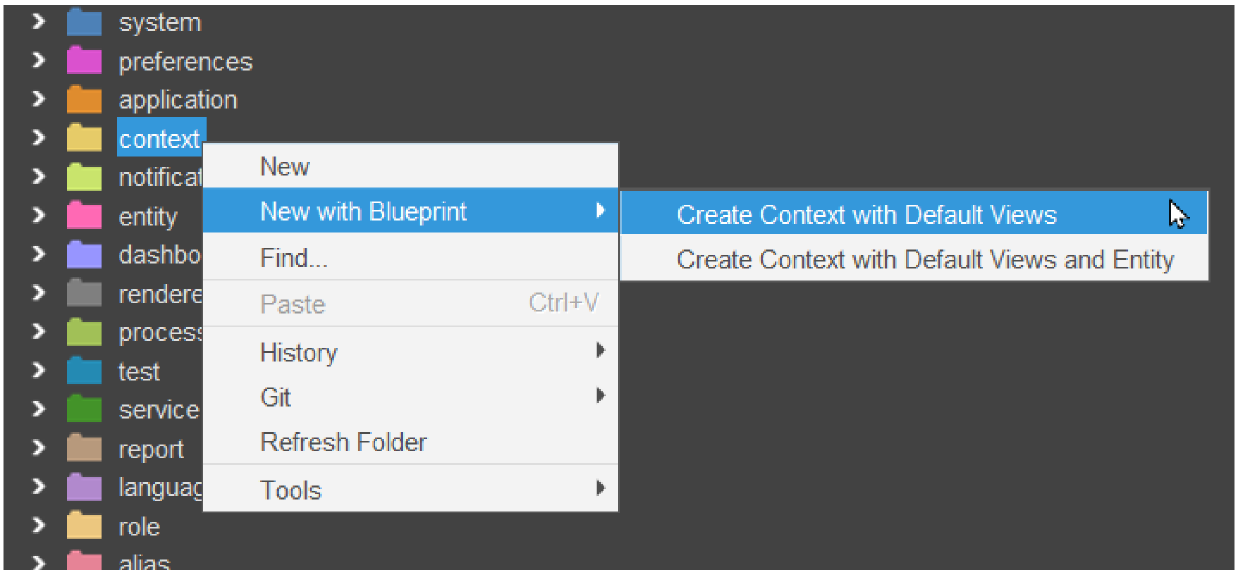


Figure 11. Blueprints available for node "context"

The Blueprints displayed in the above screenshots work as follows:

- entity > New with Blueprint > Generate Entity from database: Generates a new Entity with EntityFields (including correct contentType) and RecordContainer connection, based on an existing database table.
- context > New with Blueprint > Create Context with default Views: Generates a new Context with selectable default Views (PreviewView, MainView, etc.), based on an existing Entity.

- * context > New with Blueprint > Create Context with default Views and Entity: Generates a new Context with an Entity and selectable default Views (PreviewView, MainView, etc.). In this case, an existing database table is not required, and you need to create the RecordContainer by yourself.

Thus, in most cases, your usual approach will not be to create Context, Entity, EntityFields, and RecordContainer manually step-by-step, as explained in previous chapters; but you will simplify your work by first creating a database table and then executing the Blueprints "Generate Entity from database" and "Create Context with default Views" subsequently. Nevertheless, this automation still requires some subsequent work, e.g., configuring ViewTemplates for the Views.

Find further information about available Blueprints in ADITO Information Document AID114 "Blueprints". This document also explains how to create additional Blueprints, according to your own requirements.

9.3. Extend the Global Menu

The last step to make our data visible, is to extend our frontend’s "Global Menu" by a new menu group, including 3 new menu item, one for each Context: Double-click on _SYSTEM_APPLICATION_NEON (in the "Projects" menu, under "application"). This will open an editor window in the upper middle part of the Designer, showing the current menu items (on the left) and all existing ADITO elements (on the right) that, in principle, are suitable for being connected via a menu entry. As you can see, a menu item never stands alone, but all menu items belong to a menu group, e.g., "Contact Management", or "Sales".

You can filter the elements to be shown by setting the checkboxes in the Navigator window. In our case, we set the checkbox at "NeonContext", in order to restrict the displayed elements to Contexts.

Let’s start with the Context Car: Place the Context Car somewhere on an empty space, e.g., under the group "Contact Management", by simply dragging it from the right to the left, and dropping it in a dark grey area. Automatically, a new menu group named "Group" will be created, with an (unnamed) sub-group in it, which in turn includes the menu item "Car". Click on the default name "Group" and change its property "title" to a suitable term, e.g., "Car Pool". Now, also add menu items for the Contexts CarDriver and CarReservation, by dragging and dropping the Contexts inside (!) the new Carpool sub(!)-group.

As you can see, the Contexts' menu items receive the Contexts' names by default. If required, you could overwrite this name by setting the menu item’s property "title".

A menu item is generally not visible unless a project role has been assigned to the sub-group it’s residing in. This allows you to restrict specific menu items to specific user groups. As the test user "Admin" (included in the ADITO xRM project) already has the project role

"INTERNAL_ADMINISTRATOR", we will assign this role to the sub-group including the new menu items: In the Navigator window, set the checkbox "Role"; this restricts the display in the editor window to the user roles. Then, drag the role "INTERNAL_ADMINISTRATOR" and drop it inside the sub-group. A small orange square is shown in the upper right corner of the sub-group's title bar, indicating that the role is assigned. If you click on this orange square, all assigned roles are shown.

The role "INTERNAL_ADMINISTRATOR" as well as further roles with prefix "INTERNAL_" are *internal* roles that can neither be extended nor deleted. Nevertheless you can create your own *project* roles (see sub-chapter below).

> Please always be aware that the only purpose of this kind of role assignment is to control the Global Menu, in a user-specific way. BUT if a user knows the URL of a Context, he can open it even if he does not see it in the Global Menu. If you want to restrict specific Contexts, tabs etc. to specific user groups, you can do this via custom roles, to be defined in the client, in menu group "User Administration". Find more information in the ADITO Information Document Roles and Access Rights.

9.3.1. Creating new project roles

To create a new project role,

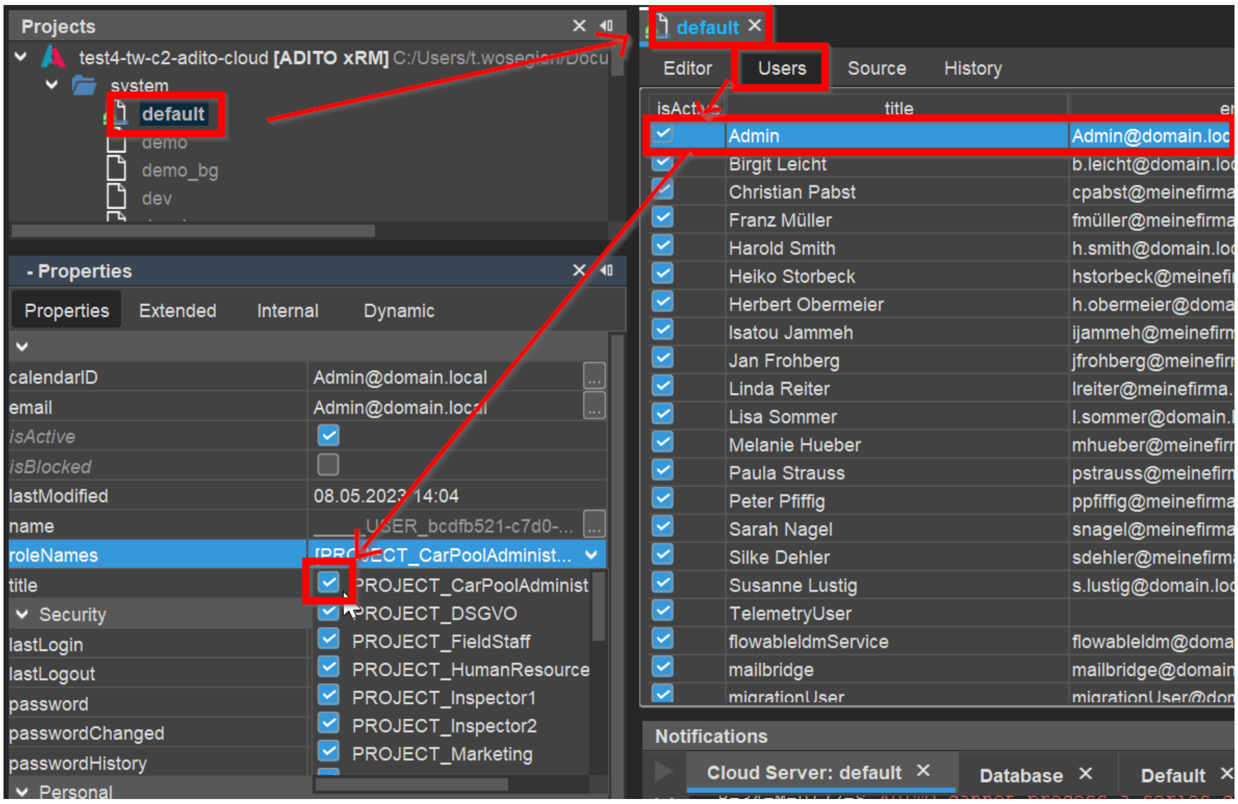
- navigate to folder "role" in the "Projects" window (project tree).
- Right-click on the folder and choose "New" from the context menu.
- Enter a name in CamelCase, with the prefix "PROJECT_", i.e., "PROJECT_CarPoolAdministrator".
- The role will immediately be visible in the role assignment menu (application > ____SYSTEM_APPLICATION_NEON)

Now we can assign the new role "PROJECT_CarPoolAdministrator" to the menu sub-group that holds the Carpool-related Contexts (by drag-and-drop, see above). For testing purposes, we can remove the role assignment INTERNAL_ADMINISTRATOR, by clicking on the small orange square and then on the cross icon to the right of INTERNAL_ADMINISTRATOR.

When we now deploy, logout and login again, our Carpool-related Contexts have vanished from the Global Menu. The reason is that, so far, we have not yet assigned the new role to a user. This can be done quite quickly:

- In the project tree, double-click on system > default.
- Click on tab "Users" in the middle part of the Designer.
- Click on line "Admin" (because this is our test user, with whom we log into the client)

- Check the new role in property "roleNames"
- Save


Projects

test4-tw-c2-adito-cloud [ADITO xRM] C:/Users/t.wosegien/Docu

system

default

demo

demo_bg

dev

- Properties

Properties | Extended | Internal | Dynamic

calendarID	Admin@domain.local
email	Admin@domain.local
isActive	✓
isBlocked	
lastModified	08.05.2023 14:04
name	____USER_bcdfb521-c7d0-...
roleNames	[PROJECT_CarPoolAdminist...
title	✓ PROJECT_CarPoolAdminist
Security	✓ PROJECT_DSGVO
lastLogin	✓ PROJECT_FieldStaff
lastLogout	✓ PROJECT_HumanResource
password	✓ PROJECT_Inspector1
passwordChanged	✓ PROJECT_Inspector2
passwordHistory	✓ PROJECT_Marketing
Personal	

default

Editor | Users | Source | History

isActive	title	em
✓	Admin	Admin@domain.loc
✓	Birgit Leicht	b.leicht@domain.loc
✓	Christian Pabst	cpabst@meinefirma
✓	Franz Müller	fmüller@meinefirma
✓	Harold Smith	h.smith@domain.loc
✓	Heiko Storbeck	hstorbeck@meinefi
✓	Herbert Obermeier	h.obermeier@doma
✓	Isatou Jammeh	ijammeh@meinefirm
✓	Jan Frohberg	jfrohberg@meinefir
✓	Linda Reiter	lreiter@meinefirma.
✓	Lisa Sommer	l.sommer@domain.l
✓	Melanie Hueber	mhueber@meinefirr
✓	Paula Strauss	pstrauss@meinefirr
✓	Peter Pfiffig	ppfiffig@meinefirma
✓	Sarah Nagel	snagel@meinefirma
✓	Silke Dehler	sdehler@meinefirm
✓	Susanne Lustig	s.lustig@domain.loc
✓	TelemetryUser	
✓	flowableldmService	flowableldm@doma
✓	mailbridge	mailbridge@domain
✓	migrationUser	migrationUser@dom

Notifications

Cloud Server: default | Database | Default

If you now deploy (see chapter below), logout and login again, you should see the carpool-related Contexts again in the Global Menu.

9.4. Deploy

To convert our additions and changes into a form and structure that can be "understood" by the ADITO server, we need to execute a deploy process (commonly only called "deploy")

9.4.1. Practically

Make sure that the ADITO server is running and there is a tunneled database connection (see above). Then, click on the "Deploy Project" button. This button, which shows a burger-like icon (with a blue arrow from the left), is located in the middle of the Designer’s button bar:

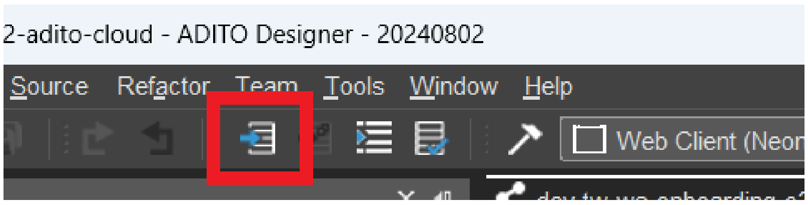

2-adito-cloud - ADITO Designer - 20240802

Figure 12. The "Deploy project" button

When this button has been clicked, a deploy dialog will open, in which you select your project (e.g., "dev-mycloudsystem-c2-adito-cloud") and your database alias (e.g., "default"). As for the checkboxes, you can leave the default settings. Confirm with "OK".

Then, at first, so-called "Transpile" is executed. This may take some time, especially if you deploy for the first time. The technical term "transpiling" derives from modularization, which is explained in the ADITO Information Document AID123 Modularization.

After the transpile is finished, the actual deploy starts with a comparison between the code in the Designer and the state of the cloud system. (This also may take some time, especially if you deploy for the first time.)

Then, a dialog will open subsequently, showing all models of the project that include changes, preceded by the following "SQL-like" abbreviations:
* "I" stands for "Insert", meaning models that you have recently created/added to your project.
* "U" stands for "Update", meaning existing models that you have recently changed (e.g., you added a new menu group in the Global Menu, as we have just done it - see above).
* "D" stands for "Delete", meaning existing models that you have recently deleted.

If you execute the very first deploy to a cloud system, a large number of changed models might be shown. This has technical reasons, which are not explained here. Just deploy all of them, then they will not appear again with the next deploy.

If you want to deploy these changes, you can leave all checkboxes checked and confirm with "OK". After a few seconds, a dialog stating "n model(s) were updated" appears in the lower right corner of the Designer.

Every development and configurations you perform with the Designer will only be visible in the client if you deploy them first. Please consider this when reading the following chapters, as usually there will not always be reminders to deploy. After the deploy is completed, you need to refresh the view in the browser. As clicking the browser’s "refresh" button is sometimes not enough, we recommend to click on the respective menu item instead. In cases when menu items have been changed, you need to log out and log into the client again, in order to see the changes. In rare cases, and depending on the browser you use, it can also be necessary to empty the browser’s cache - in Google chrome, e.g., via the shortcuts CTRL+F5 ("deep refresh" of the current web page) or CTRL+SHIFT+DEL > "Clear data" (deleting selected cache data) - in order to see the changes effected by the deploy. Last but not least, changes of a few system-related properties require a restart of the server. Please find more information in chapter Troubleshooting.

Always make sure you had actually executed a deploy, whenever you wonder why a change is not visible in the client.

Deploy of a single model

If you have changed only one single ADITO model (e.g., its property "title"), you can save time if you do not execute the deploy for the complete project, but restrict it to only the respective ADITO model (e.g., the Context, the Entity, or the View). You have the following options:

- In the project tree, right-click on the model that you have changed (e.g., the Context, the Entity, or the View), and choose "Deploy" from the context menu.
- If you have opened the model as tab in the Editor (upper middle part of the Designer), you can also right-click on the tab and choose "Deploy". If the tab shows a process, the deploy will nevertheless be performed for the complete model related to the process. Thus, for example, if you have changed 2 processes of the same Entity, a deploy performed via the tab of one process will also deploy the other process.
- If you deploy a single Context, all elements that appear subordinated to it in the project tree (its Entity and Views) will NOT automatically be deployed, too.

9.4.2. Technically

To understand what happens technically, if you deploy, you first have to understand how an ADITO project is stored:

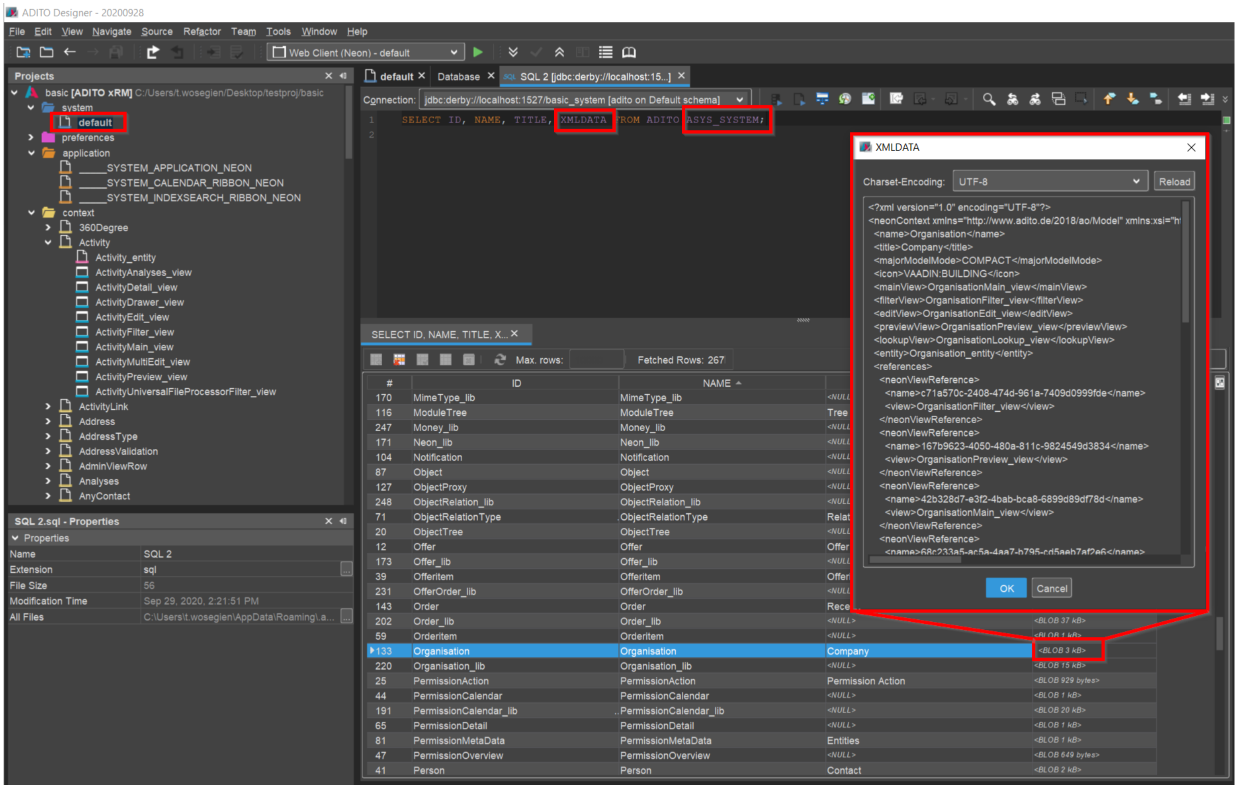
All configuration and code of an ADITO project is stored in, basically, 3 types of files:

- .aod files include the configuration of an ADITO model (e.g., an Entity) in XML form. For example, if you double-click on an Entity in the "Projects" window, you can inspect its XML data by clicking on tab "Source" in the Editor (upper middle part of the Designer). If you hover over the tab title with your mouse pointer, a tooltip is displayed, indicating the file name (e.g., Activity_entity.aod) and its directory path.
- .js files include the **J**ava**S**cript/JDito code entered in properties or libraries (e.g., a valueProcess).
- Files named documentation.adoc include the content of a property "documentation", which has "AsciiDoc" format, see https://asciidoc.org/

Now, technically, a "deploy" means that the ADITO project's source data (.aod and .js files) are written into the system database, in particular, into the table ASYS_SYSTEM. The ADITO web server then reads this table's content in order to create web pages. If you choose option "Force deploy" when executing a deploy, then all datasets of table ASYS_SYSTEM are deleted, and the *complete* project data is inserted anew. This can be necessary in rare cases, when the Designer fails to recognize certain changes you had performed in your project.

If you want to inspect the content of table ASYS_SYSTEM, you can proceed as follows:

- In the "Projects" window, double-click on system > default.
- In the Editor (upper middle part of the designer), double-click on "____SYSTEMALIAS".
- A database tree appears. Here, navigate to "adito_system" > "Tables" > "Tables" > ASYS_SYSTEM.
- Right-click on ASYS_SYSTEM and choose "View Data ..." from the context menu.
- Now you see all (or the first 100) datasets of this table. In particular, look at the following columns:
    - NAME indicates the name of the corresponding ADITO model.
    - XMLDATA includes the ADITO model's XML or JavaScript/JDito content. You can inspect this content by double-clicking on one XMLDATA cell: A menu appears, in which you choose option "Open as text". Then you see the same content that you can see in the "Source" tab of the Editor, after double-clicking on an ADTIO model in the "Projects" window (e.g., an Entity).



SQL 2.sql - Properties

Name	SQL 2
Extension	sql
File Size	56
Modification Time	Sep 29, 2020, 2:21:51 PM
All Files	C:\Users\t.wosegien\AppData\Roaming\.a...

Connection: jdbc:derby://localhost:1527/basic_system [adito on Default schema]

```
SELECT ID, NAME, TITLE, XMLDATA FROM ADITO ASYS_SYSTEM;
```

Charset-Encoding: UTF-8

```
<?xml version="1.0" encoding="UTF-8"?>
<neonContext xmlns="http://www.adito.de/2018/ao/Model" xmlns:xsi="ht
  <name>Organisation</name>
  <title>Company</title>
  <majorModelMode>COMPACT</majorModelMode>
  <icon>VAADIN:BUILDING</icon>
  <mainView>OrganisationMain_view</mainView>
  <filterView>OrganisationFilter_view</filterView>
  <editView>OrganisationEdit_view</editView>
  <previewView>OrganisationPreview_view</previewView>
  <lookupView>OrganisationLookup_view</lookupView>
  <entity>Organisation_entity</entity>
  <references>
    <neonViewReference>
      <name>c71a570c-2408-474d-961a-7409d0999fde</name>
      <view>OrganisationFilter_view</view>
    </neonViewReference>
    <neonViewReference>
      <name>167b9623-4050-480a-811c-9824549d3834</name>
      <view>OrganisationPreview_view</view>
    </neonViewReference>
    <neonViewReference>
      <name>42b328d7-e3f2-4bab-bca8-6899d89df78d</name>
      <view>OrganisationMain_view</view>
    </neonViewReference>
    <neonViewReference>
      <name>68c233a5-ac5a-4aa7-b795-cd5aeb7af2e6</name>
```

#	ID	NAME		
170	MimeType_lib	MimeType_lib	<NULL	
116	ModuleTree	ModuleTree	Tree	
247	Money_lib	Money_lib	<NULL	
171	Neon_lib	Neon_lib	<NULL	
104	Notification	Notification	<NULL	
87	Object	Object	<NULL	
127	ObjectProxy	ObjectProxy	<NULL	
248	ObjectRelation_lib	ObjectRelation_lib	<NULL	
71	ObjectRelationType	ObjectRelationType	Relat	
20	ObjectTree	ObjectTree	<NULL	
12	Offer	Offer	Offer	
173	Offer_lib	Offer_lib	<NULL	
39	Offeritem	Offeritem	Offer	
231	OfferOrder_lib	OfferOrder_lib	<NULL	
143	Order	Order	Rece	
202	Order_lib	Order_lib	<NULL>	<BLOB 37 kB>
59	Orderitem	Orderitem	<NULL>	<BLOB 1 kB>
133	Organisation	Organisation	Company	<BLOB 3 kB>
220	Organisation_lib	Organisation_lib	<NULL>	<BLOB 15 kB>
25	PermissionAction	PermissionAction	Permission Action	<BLOB 929 bytes>
44	PermissionCalendar	PermissionCalendar	<NULL>	<BLOB 1 kB>
191	PermissionCalendar_lib	PermissionCalendar_lib	<NULL>	<BLOB 20 kB>
65	PermissionDetail	PermissionDetail	<NULL>	<BLOB 1 kB>
81	PermissionMetaData	PermissionMetaData	Entities	<BLOB 1 kB>
47	PermissionOverview	PermissionOverview	<NULL>	<BLOB 649 bytes>
41	Person	Person	Contact	<BLOB 2 kB>

Figure 13. Inspecting the content of the system table ASYS_SYSTEM

9.5. A first test

Now it's time to watch our interim results in the web client, i.e., in the browser: Make sure that

1. the ADITO server is running,
2. the tunnels have been established,
3. you have deployed all changes (see above),

and then call the URL of your cloud system. Instead of doing this manually, you can also select "Web Client (Neon) Cloud - default" in the combo box of the Designer’s button bar, and click on the green triangle to the right of the combo box. (Due to a bug, the URL that is now generated includes the port number, which leads to an error message. Please remove the port manually, including the colon (e.g. ":8080") and call the URL again.) Then, a login mask appears, where you enter "admin" as username and the admin password, and confirm by "Login". The admin password is available in the SSP (see chapter ADITO server).

After a few seconds, the Dashboard appears (we will deal with the handling of Dashboards later). Click on the little blue text in the upper middle (usually the word "Home", along with a "house" icon), which will show you all menu items, including the items we have just added.

If you click on menu item "Car", the FilterView of the Context Car is displayed (this is due to ADITO convention). Here, you see only an empty table.

As you will have noticed, menu group "Car Pool" is currently displayed with a question mark to the left of it. If you click on this menu group, you will see further questions marks in the vertical button bar on the left (available via "burger button" in the left upper corner of the client). The placeholder "?" indicates, that there is no icon specified yet. You can find information about how to search and integrate a suitable icon in chapter "Icons", subchapter of Controlling the design.

9.5.1. Entering example data

For testing purposes, it can be helpful to have some example data available. As we have not configured all EntityFields yet, we cannot enter data via the client. Thus, you must either enter data directly, using ADITO’s database editor (system > default > Data_alias > ADITO), or you can again use Liquibase. The latter works as follows:

In folder example_carpool (under alias > Data_alias), create an empty xml file and name it "init_car.xml". Open it in ADITO and fill in the following code:

init_car.xml

```
<?xml version="1.1" encoding="UTF-8" standalone="no"?>
<databaseChangeLog xmlns="http://www.liquibase.org/xml/ns/dbchangelog" xmlns:ext="http://www.liquibase.org/xml/ns/dbchangelog-ext"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog-ext
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-ext.xsd http://www.liquibase.org/xml/ns/dbchangelog
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.6.xsd">
    <changeSet author="j.smith" id="9bb1c6b9-430a-4ba4-90ad-cbde9ae38660">
        <insert tableName="CAR">
```

```
            <column name="CARID" value="405de82e-4324-47d7-a643-d66ed1b4ea77"/>
            <column name="COLOR" value="RED"/>
            <column name="LICENSEPLATENUMBER" value="LA-AD 123"/>
            <column name="MANUFACTURER" value="BMW"/>
            <column name="MANUFACTUREDATE" valueDate="2022-11-21"/>
            <column name="TYPE" value="320i"/>
            <column name="PRICE" valueNumeric="34532.52"/>
            <column name="CURRENCY" value="EUR"/>
        </insert>
        <insert tableName="CAR">
            <column name="CARID" value="324445c7-57e0-4e26-b83f-beece3b42a2d"/>
            <column name="COLOR" value="GREEN"/>
            <column name="LICENSEPLATENUMBER" value="M-CX 9876"/>
            <column name="MANUFACTURER" value="MERCEDES"/>
            <column name="MANUFACTUREDATE" valueDate="2020-03-09"/>
            <column name="TYPE" value="C220"/>
            <column name="PRICE" valueNumeric="42934.16"/>
            <column name="CURRENCY" value="USD"/>
        </insert>
        <insert tableName="CAR">
            <column name="CARID" value="84b5ac0a-f490-4537-b4a9-273279f01319"/>
            <column name="COLOR" value="YELLOW                              "/>
            <column name="LICENSEPLATENUMBER" value="H-LK 597"/>
            <column name="MANUFACTURER" value="FORD"/>
            <column name="MANUFACTUREDATE" valueDate="2021-01-11"/>
            <column name="TYPE" value="Focus"/>
            <column name="PRICE" valueNumeric="23934.16"/>
            <column name="CURRENCY" value="USD"/>
        </insert>
    </changeSet>
</databaseChangeLog>
```

The "column name" tags include different data types, such as "value", "valueNumeric", and "valueDate". Make sure you are always using a data type fitting to the data type of the corresponding database column. Find further information in the chapter on Liquibase in the Designer Manual.

This new init_car.xml file must now be referenced in the changelog.xml of our car pool project (under alias > Data_alias > example_carpool) by adding the following additional code line:

changelog.xml (under alias > Data_alias > example_carpool)

```
...
<include relativeToChangelogFile="true" file="init_car.xml"/>
...
```

Now, execute a Liquibase update (see previous chapter) and refresh the display by clicking again on the globe icon and selecting the menu item "Car" (for technical reasons, using the browser's refresh button is not enough). Then, the inserted example datasets should be displayed.

Of course, our application is still far from being completed. E.g., you see the 36-digit ID of a color (e.g., "RED ") instead of the name of the color (e.g., "red"), you cannot select a car driver from a list of employees, and there is no relation between the Contexts Car, CarDriver, and CarReservation. But, for example, you can already create a new dataset (including part of the fields) or edit an existing dataset. Or you can load and save an image of the car: Click on the "pencil" icon to the right of the image's placeholder icon (upper left part of the PreviewView), then on the placehoder itself, select an image stored on your computer, and save the selection by clicking on the blue hooklet.

If you want to have your own example data available in your system, there are 2 alternatives to the above procedure:

- You fill your database tables directly via SQL "INSERT" scripts: Press the "Execute SQL" button to the left of the combo box in the button bar, select your connection, insert your SQL code, and run it (F6 or button "Run SQL" to the right of the "Connection" combo box).
- You enter example data via the EditViews in the ADITO client.

In both cases, it might be useful to auto-generate Liquibase "init" files afterwards, because then, e.g., you can reset your database via the Liquibase function "Drop all & Update…" and still have the example data present. Please refer to chapter "Create Liquibase files automatically" of the Designer Manual.

9.6. Dashboard and Dashlet

After you have logged in to the client or whenever you click the "Home" button, a Dashboard appears. This is called the "Home" Dashboard.

A Dashboard basicly consists of one or more View components, the so-called Dashlets. Via the "Dashlets" button (lower right part of a Dashboard), you can open the so-called DashletStore: Here, you can add further Dashlets by selecting them from a category list. To remove a Dashlet, click on its "x" button (upper right corner of the Dashlet).

9.6.1. Add Dashlets

To make a specific View available as Dashlet in the DashletStore, we first create a so-called DashletConfiguration (a kind of template) and assign it to the View:
Open the View in the Navigator window (let's use CarReservationFilter_view as example) and choose "Add Dashlet Config" from its context menu.

This option is not available for Views having a "MasterDetailLayout".

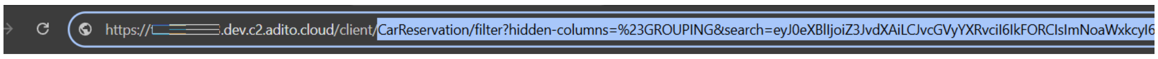
Enter the name of the DashletConfiguration according to the ADITO spelling guidelines (see ADITO Information Document AID001, chapter "Spelling & Wording" > "ADITO models"), e.g., "AllCarReservations". The new DashletConfiguration's name appears under the node "DashletConfigs". Edit the DashletConfiguration's properties:

- title: Title of the Dashlet, to be visible in the client, when the Dashlet has been added to the Dashboard, e.g., "Car reservations"

- description: Description of the Dashlet, e.g. "Show all car reservations". This text will be visible in the DashletStore, below the Dashlet’s title.
- icon: Mandatory property. Every Dashlet needs an icon to be set, in order to ensure a good identification. If no icon is set, you get an error message.

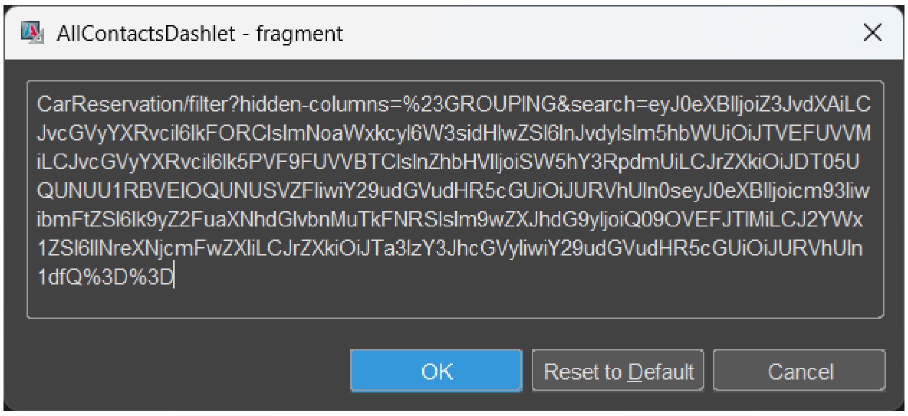
  > The icon must be set, otherwise the DashletStore won’t work properly.

- fragment: The last part of the View’s URL (when opened via a Context, not via a Dashboard), following "/client/"). E.g., if the View’s URL is https://myProject.dev.c2.adito.cloud/client/CarReservation/filter, you must enter "CarReservation/filter", if you want to see all reservations, unfiltered.
  If you want to apply a filter, you can simply extend this fragment by "?search=…": Just configure the filter in the FilterView, apply it, and then copy the last part of the URL into the property "fragment" (it will be a long cryptic URL):

  ```
  https://[redacted].dev.c2.adito.cloud/client/CarReservation/filter?hidden-columns=%23GROUPING&search=eyJ0eXBlIjoiZ3JvdXAiLCJvcGVyYXRvciI6IkFORCIsImNoaWxkcyI6
  ```

  →

  AllContactsDashlet - fragment

  ```
  CarReservation/filter?hidden-columns=%23GROUPING&search=eyJ0eXBlIjoiZ3JvdXAiLC
  JvcGVyYXRvciI6IkFORCIsImNoaWxkcyI6W3sidHlwZSI6InJvdyIsIm5hbWUiOiJTVEFUVVM
  iLCJvcGVyYXRvciI6Ik5PVF9FUVVBTCIsInZhbHVlIjoiSW5hY3RpdmUiLCJrZXkiOiJDT05U
  QUNUU1RBVEIOQUNUSVZFIiwiY29udGVudHR5cGUiOiJURVhUIn0seyJ0eXBlIjoicm93Iiw
  ibmFtZSI6Ik9yZ2FuaXNhdGlvbnMuTkFNRSIsIm9wZXJhdG9yIjoiQ09OVEFJTlMiLCJ2YWx
  1ZSI6IlNreXNjcmFwZXIiLCJrZXkiOiJTa3lzY3JhcGVyIiwiY29udGVudHR5cGUiOiJURVhUIn
  1dfQ%3D%3D
  ```

  Then, log out and log into the client. Result: Included in the Dashboard, the Dashlet immediately shows the filtered data.

- singleton: Defines, whether or not the Dashlet can be added multiple times to the Dashboard. If true, the Dashlet can only be added once. As soon as is is added, it disappears from the category list for adding new Dashlets. If false, you may add the Dashlet as often as you like.
- categories: In a configuration table, you can define, in which category in the DashletStore the Dashlet appears, when you press the "Add" button. Click the plus button ("+") to define the name and the title of a category, e.g. name = "carReservation", title = "Car reservation". The title will be visible as category in the DashletStore, the name is only for internal organisation.

Now try to create 2 DashletConfigurations: one showing all reservations (see example configuration above), and the other showing only cars with license plate numbers starting with "M" (name it "CarsLicensePlateStartingWithM"). Then, add them to the "Home" Dashboard in the client.

Note that you need to logout and re-login in order to make new or modified Dashboards/Dashlets visible in the DashletStore of the client.

9.6.2. Configure Dashboard defaults

You may have noticed that in the "Projects" window, there is a node named "dashboard", including several Dashboards, e.g.

- "Home": This is the main Dashboard shown after logging in to the client or whenever you press the "Home" button. Never rename it.
- "SalesDashboard": This is a Dashboard showing figures of sales topics. In the Global Menu of the client, it appears like a Context: You can select it in the menu group "Sales".

The main properties of a Dashboard are:

- title: Title of the Dashboard, to be visible in the client.
- icon: The icon of the Dashlet, which will be shown
  - above the Dashlet’s title in the DashletStore;
  - on the left of the Dashlet’s title bar.
- DashboardType:
  - PRIVATE: The Dashboard’s appearance is user-specific. Any changes (e.g., moving Dashlets) a user makes, are not visible for other users. Example: the "Home" Dashboard.
  - PUBLIC:
    - All users see the same number of Dashlets.
    - Every user can change the *order* of the Dashlets, i.e., move Dashlets or change their size, but *not close* them. These individual changes are not visible to other users.
    - However:

      If a PUBLIC Dashboard has its property "fixedDashlets" set to true, then Dashlets can only be changed by a user having one of the roles specified in the Dashboard’s property "editRoles" (see below), and the position and the size of all Dashlets are the same for every user.

> (Note that property "fixedDashlets" has currently only an effect on PUBLIC Dashboards.)

- editRoles: Roles of users who are entitled to edit the Dashboard (move Dashlets, add new Dashlets, close Dashlets etc.). If no role is assigned, every user can edit.

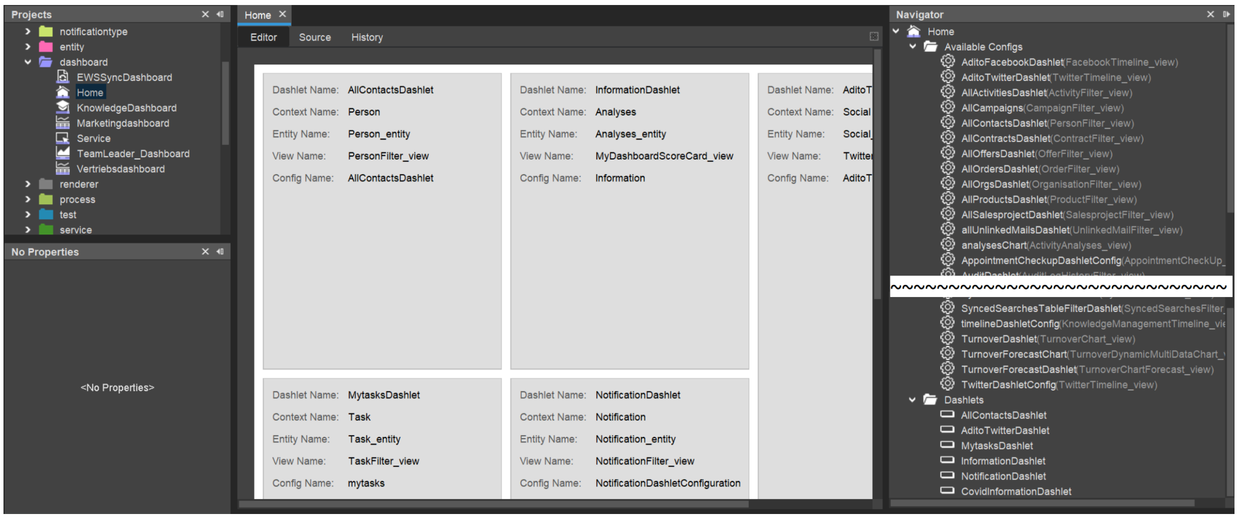
The Dashboard administrator (= a user having the role "INTERNAL_DASHBOARDSTOREADMIN") can publish, edit, and delete elements (DashletConfigurations) in the DashletStore.

In the "Projects" window, double-click on a Dashboard to view its default configuration:

In the Editor window (upper middle window of the Designer) you can see a sketch of the arrangement of the default Dashlets, i.e., their size and position.

In the Navigator window (upper right window of the Designer) you can see 2 nodes:

- Under "Dashlets" you see the default Dashlets included in the selected Dashboard (if you click on one of it, it is marked with a surrounding blue line in the Editor). You can re-order and re-size them by changing their properties (which are based on an invisble grid of columns and rows)
  - "xPos": the number of the column of the left upper corner of the Dashlet)
  - "yPos": the number of the row of the left upper corner of the Dashlet)
  - "colspan": the width of the Dashlet (= the number of rows it ranges over)
  - "rowspan": the height of the Dashlet (= the number of rows it ranges over)
- Under "Available Configs", you see all DashletConfigs of the project. To add a new Dashlet based on a specific DashletConfig, right-click on it and choose "Add to Dashboard".



Dashlet Name:	AllContactsDashlet
Context Name:	Person
Entity Name:	Person_entity
View Name:	PersonFilter_view
Config Name:	AllContactsDashlet

Dashlet Name:	InformationDashlet
Context Name:	Analyses
Entity Name:	Analyses_entity
View Name:	MyDashboardScoreCard_view
Config Name:	Information

Dashlet Name:	AditoT
Context Name:	Social
Entity Name:	Social_
View Name:	Twitte
Config Name:	AditoT

Dashlet Name:	MytasksDashlet
Context Name:	Task
Entity Name:	Task_entity
View Name:	TaskFilter_view
Config Name:	mytasks

Dashlet Name:	NotificationDashlet
Context Name:	Notification
Entity Name:	Notification_entity
View Name:	NotificationFilter_view
Config Name:	NotificationDashletConfiguration

```
Home
  Available Configs
    AditoFacebookDashlet(FacebookTimeline_view)
    AditoTwitterDashlet(TwitterTimeline_view)
    AllActivitiesDashlet(ActivityFilter_view)
    AllCampaigns(CampaignFilter_view)
    AllContactsDashlet(PersonFilter_view)
    AllContractsDashlet(ContractFilter_view)
    AllOffersDashlet(OfferFilter_view)
    AllOrdersDashlet(OrderFilter_view)
    AllOrgsDashlet(OrganisationFilter_view)
    AllProductsDashlet(ProductFilter_view)
    AllSalesprojectDashlet(SalesprojectFilter_view)
    allUnlinkedMailsDashlet(UnlinkedMailFilter_view)
    analysesChart(ActivityAnalyses_view)
    AppointmentCheckupDashletConfig(AppointmentCheckUp_
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    SyncedSearchesTableFilterDashlet(SyncedSearchesFilter_
    timelineDashletConfig(KnowledgeManagementTimeline_vie
    TurnoverDashlet(TurnoverChart_view)
    TurnoverForecastChart(TurnoverDynamicMultiDataChart_
    TurnoverForecastDashlet(TurnoverChartForecast_view)
    TwitterDashletConfig(TwitterTimeline_view)
  Dashlets
    AllContactsDashlet
    AditoTwitterDashlet
    MytasksDashlet
    InformationDashlet
    NotificationDashlet
    CovidInformationDashlet
```

Figure 14. The configuration of the "Home" Dashboard

According to the ADITO spelling guidelines (see ADITO Information Document AID001, chapter "Spelling & Wording" > "ADITO models"), the name of a Dashlet starts with the name of the assigned DashletConfiguration, followed by the suffix "Dashlet", e.g. AllContactsDashlet.

Be aware that the above configurations are only default settings. These are only applied once, when starting the client after a new ADITO installation.
Whenever the user modifies the Dashboard in the client, the default settings visible in the Designer remain unchanged.

9.6.3. Resetting Dashboards

If users want their Dashboard to be re-setted to its initial state (as configured in the Designer), the procedure is different depending on whether it is a "public" Dashboard or a "private" Dashboard:

9.6.3.1. Reset of a "public" Dashboard

To reset a "public" Dashboard (e.g., the "Sales Dashboard" of ADITO xRM), there are 2 ways:

- In the client: Remove all Dashlets manually.
- On database level: Remove all Dashlet datasets referring to the respective Dashboard, from the table ASYS_DASHLETS.

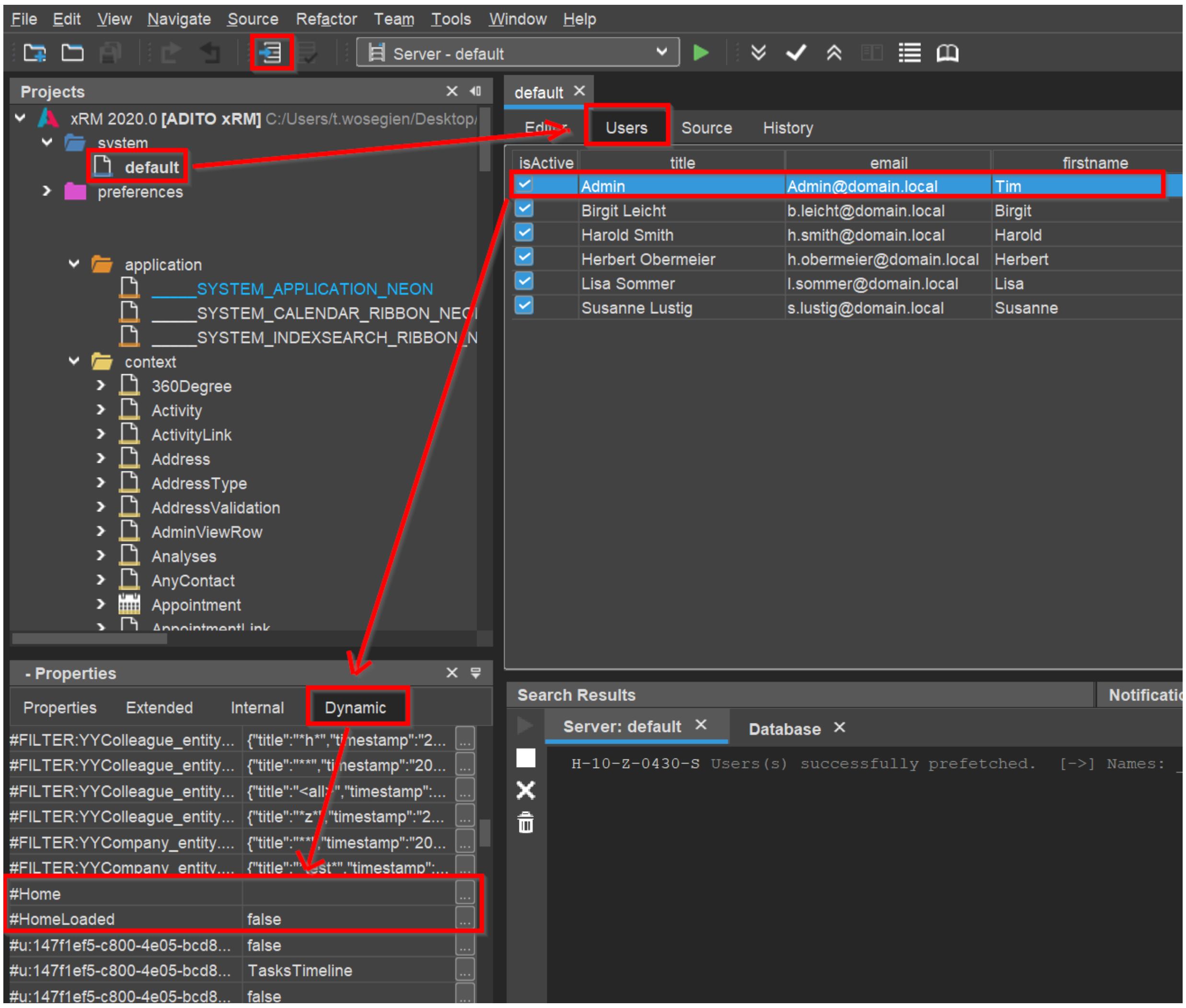
Afterwards, in both cases, re-open the Dashboard by choosing it from the Global Menu or clicking on its icon (pressing the "refresh" button of the browser is not enough). The Dashboard is now resetted.

9.6.3.2. Reset of a "private" Dashboard

To reset a "private" Dashboard (e.g., the "Home" Dashboard of ADITO xRM), there are 2 ways:

- In the **client** (currently only available for the Home Dashboard):
  - Open the DashletStore (blue button "Dashlets").
  - Click on the button `Reset Dashboard`
- In all other cases, a "private" Dashboard cannot be resetted in the client. In this case, you can reset a "private" Dashboard only in the **Designer**:
  - Open the property sheet of the respective user in the Designer:
    - In the "Projects" window, double-click on system > default
    - In the Editor window, choose tab "Users"
    - Click on the user whose Dashboard is to be resetted

- In the "Properties" window,
  - choose tab "Dynamic" (this tab exists only for users who had logged-in at least once before); a key-value list appears.
  - completely delete the content of the value field of key "#<name of Dashboard>", e.g., "#Home" for the "Home" Dashboard;
  - set the value of key "#<name of Dashboard>Loaded" (e.g., "#HomeLoaded"), to "false" (type the word `false` as value).
- Click the "Save all" button in the button bar of the Designer
- In the client, log out and log in again, and re-open the Dashboard: The Dashboard is now resetted.



isActive	title	email	firstname
✓	Admin	Admin@domain.local	Tim
✓	Birgit Leicht	b.leicht@domain.local	Birgit
✓	Harold Smith	h.smith@domain.local	Harold
✓	Herbert Obermeier	h.obermeier@domain.local	Herbert
✓	Lisa Sommer	l.sommer@domain.local	Lisa
✓	Susanne Lustig	s.lustig@domain.local	Susanne

#FILTER:YYColleague_entity...	{"title":"*h*","timestamp":"2...
#FILTER:YYColleague_entity...	{"title":"**","timestamp":"20...
#FILTER:YYColleague_entity...	{"title":"<all>","timestamp":...
#FILTER:YYColleague_entity...	{"title":"*z*","timestamp":"2...
#FILTER:YYCompany_entity....	{"title":"**","timestamp":"20...
#FILTER:YYCompany_entity....	{"title":"*test*","timestamp":...
#Home	
#HomeLoaded	false
#u:147f1ef5-c800-4e05-bcd8...	false
#u:147f1ef5-c800-4e05-bcd8...	TasksTimeline
#u:147f1ef5-c800-4e05-bcd8...	false

```
H-10-Z-0430-S Users(s) successfully prefetched.  [->] Names: _
```

Figure 15. Resetting the "private" Dashboard of a specific user

9.6.4. Creating new Dashboards

To create a new Dashboard, right-click on node "dashboard" in the Projects window and then choose "New" from the context menu. Enter the Dashboard's name, according to the ADITO spelling guidelines (see ADITO Information Document AID001, chapter "Spelling & Wording" > "ADITO models"), e.g. "CarDashboard". Configure the Dashboard's properties (see previous chapter).

Now, add Dashlets to the Dashboard form:

- In the Projects window, Double-click on your new Dashboard.
- In the Navigator Window, search for your Dashlet under the node "Available Configs", e.g., "AllCarReservations"
- Right-click on this Dashlet and choose "Add to Dashboard" from the context menu. In the Designer's Editor, the Dashlet now appears in the Dashboard form and in the Navigator window, under the node "Dashlets".
- In the Navigator window, rename the new Dashlet's default name, e.g., to "AllCarReservationsDashlet" (right-click on it and choose "Rename" from the context menu).
- Configure the Dashlet's properties (see previous chapter).
- Repeat this step for the second Dashlet, e.g., "CarsLicensePlateStartingWithM"

If, in the Dashboard form, the second Dashlet is not visible after adding it, it has been added "under" the previous Dashlet. Then simply click on the second Dashlet in the Navigator window (under "Dashlets") and change its properties "xPos"/"yPos"; then it will change its position and be visible beside the first Dashlet.

Afterwards, in the "Projects" window, double-click on application > _SYSTEM_APPLICATION_NEON to open the menu editor. Check "NeonDashboard" in the Navigator window, which will reduce the components visible in the middle window to only Dashboards. Drag the new Dashboard and drop it on a suitable place in the menu, e.g., directly above the menu entry "Car".

After deploying and logout/login you can open the new Dashboard via the Global Menu of the client.

10. Advanced functionality

In principle, you have now learned how to set up an ADITO application with the option to create, view, and edit datasets with simple fields of different data types. However, of course, we need some advanced functionality in order to build a professional application. This will be explained in the next chapters.

10.1. Consumer and Provider: Connecting Entities

According to the data model we have built before, there is a relationship between the Entities representing cars, drivers, and reservations: One car reservation is related to one car and one car driver. In terms of database structure, this means that the primary keys CAR.CARID and CARDRIVER.CARDRIVERID appear as foreign keys in the table CARRESERVATION (columns CAR_ID and CARDRIVER_ID).

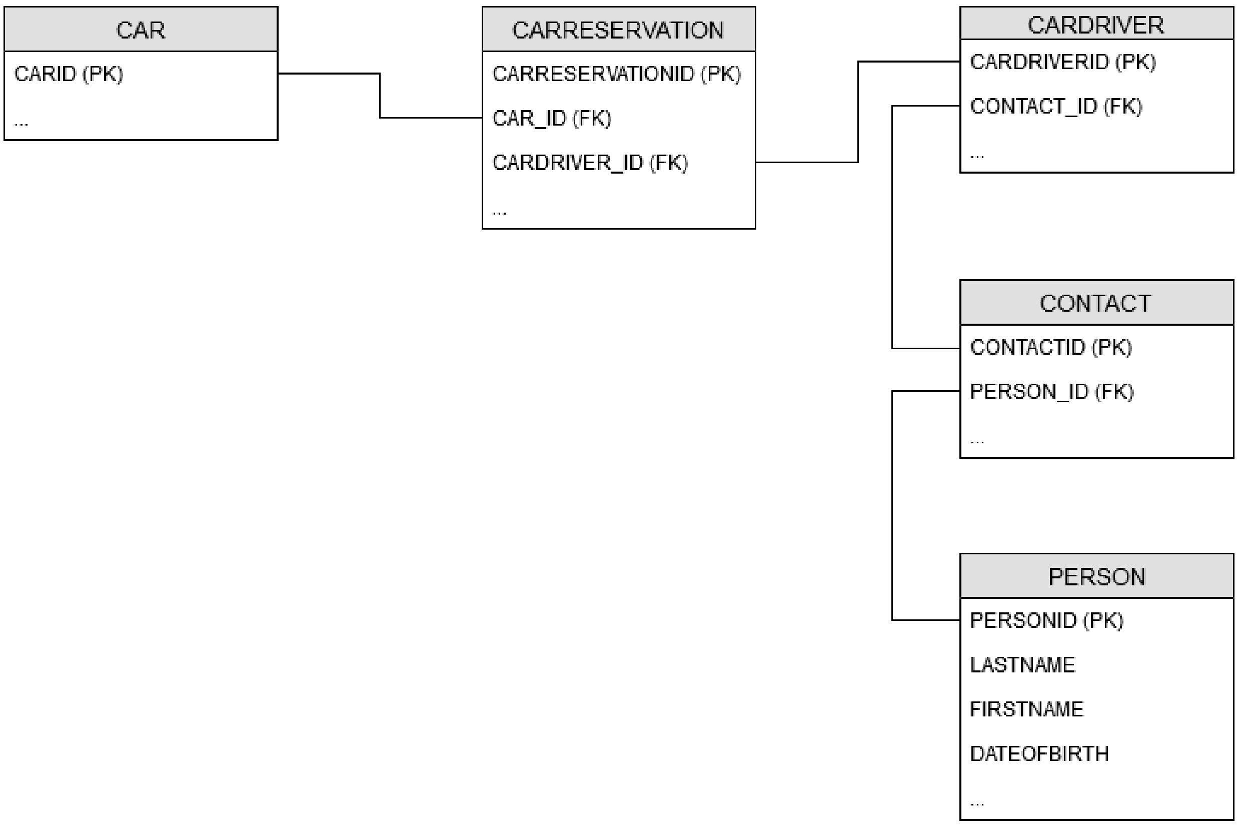


Figure 16. Carpool-related database tables with primary keys (PK) and foreign keys (FK)

Now, if we are viewing the CarReservation Context and want to create a new CARRESERVATION dataset, we must select one of the existing CARDRIVER datasets and one of the existing CAR datasets. To have all drivers and all cars available in the CarReservation Context, we need to establish dependencies between the corresponding Entities. In ADITO, this can be achieved by generating objects named Consumers and Providers, which fit to each other by specific properties and by an optional parameter.

- The **Provider** is a configuration model created at the side of an Entity providing ("sending") data requested by another Entity. (Colloquially, Designer users sometimes speak of the providing Entity itself as "the Provider" - do not mistake these terms.).
- The **Consumer** is a configuration model created at the side of an Entity consuming ("requesting") data of another Entity. (Colloquially, Designer users sometimes speak of the consuming Entity itself as "the Consumer" - do not mistake these terms.)
- The **Parameter** basically contains the criteria the Provider needs to select the required data. It is created on the Provider’s side and assigned both to Provider and Consumer. If *all* datasets of an Entity are to be provided, a Parameter is not required.

The ADITO-specific terms "Consumer", "Provider", and "Parameter" are spelled with a capital letter, in order to distinguish them from the colloquial terms "consumer", "provider", and "parameter". The same wording also applies for other ADITO-specific terms that have equivalents in colloquial language, such as "Entity" or "Context". Find more information on ADITO-specific wording in the ADITO Wording_Guideline (AID002), available in the customer area of the ADITO web site.

The following sketch illustrates this mechanism, using an existing dependency of the xRM project as example: All persons related to a specific organisation (company, club, etc.) are to be displayed in the Context named "Organisation" (in the Global Menu of the web client, it is visible as "Company").

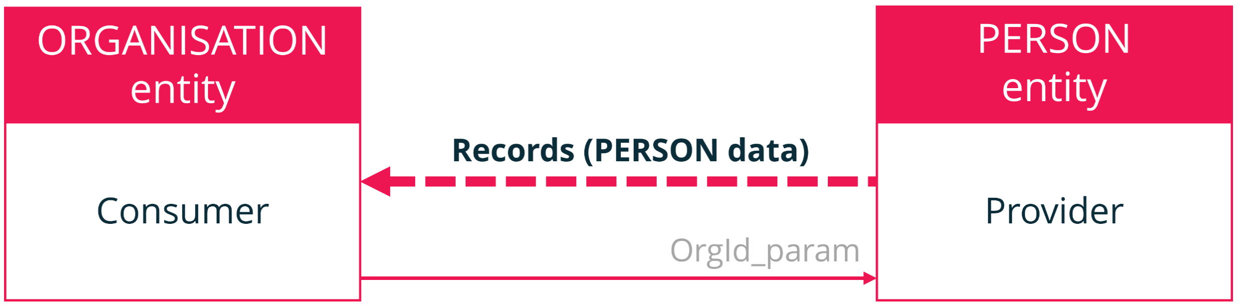


Figure 17. Example of a dependency created by Consumer and Provider

Explanations:

1. The Provider exposes one or more Parameters from its Entity to the Consumer.
2. The Consumer writes the respective value(s) (controlled by a valueProcess) into the Parameter(s).
3. The Parameters’ values are now available in the Entity of the Provider.
4. The Consumer requests the records from the Provider. As a result the recordContainer selects the new record set with the Parameter(s) being evaluated by a conditionProcess.

5. The Provider delivers the requested records to the Consumer.

This to-do list summarizes the workflow:

Table 2. Configuration of dependency via Provider and Consumer (workflow up to down)

Records required	**<Provider>_entity**	**<Consumer>_entity**
All datasets	Create new Provider: Navigator > <Provider_entity> **New Provider**: **name**: <ProviderName>	
		Create new Consumer: Navigator > <Consumer_entity> **New Consumer**: **name**: <ConsumerName> **entityName**: <Provider>_entity **fieldName**: <ProviderName>
		<ConsumingField>.consumer: <ConsumerName>
Selected datasets	Perform above steps.	
	Create new Parameter: Navigator > <Provider>_entity > **New Parameter**: **name**: <ConsumingField_param> **expose**: true	
		<ConsumingField_param>.valueProcess: `result.string(`<selection criteria>`);`
	<recordContainer>.conditionProcess: `result.string(`<WHERE condition, using `"$param.`<ConsumingField_param>`");`	

The application of this to-do list in the following chapters will make it even clearer how to create dependencies using the Provider-Consumer mechanism.

10.1.1. Example: Cars and car drivers in car reservations

In our case, in order to create a new CARRESERVATION dataset, Car_entity needs to provide all car data to CarReservation_entity, and CarDriver_entity needs to provide all car driver data to CarReservation_entity.

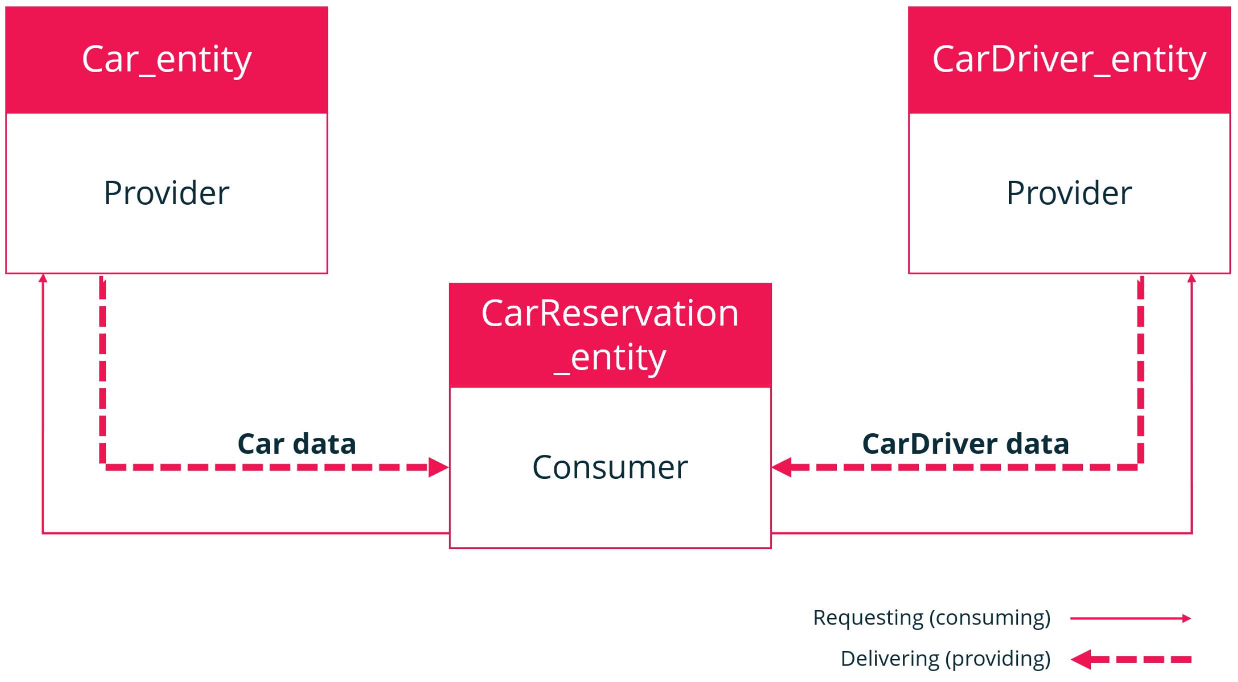


Figure 18. Dependencies in carpool example

To achieve this, we proceed as follows:

First, make sure that you have created the fields CAR_ID and CARDRIVER_ID for CarReservation_entity.

Then, in the "Projects" window, double-click on Car_entity. In the Navigator window, right-click on Car_entity and select option "New Provider" from the context menu. This will open the dialog "Create New Provider". Here, name the new Provider "Cars" (Providers are always spelled in CamelCase; if they provide multiple datasets (which is mostly the case), they are marked with plural names, see ADITO Information Document AID001, chapter "Spelling & Wording" > "ADITO models") and confirm with "OK". The new Provider will appear under the node "Providers" (Navigator window).

Next, open CarReservation_entity in the Navigator window and select option "New Consumer" from the context menu. Name the new Consumer also "Cars". Next, edit the Consumer’s properties: Select "Car_entity" as "entityName", and "Cars" as "fieldName" (= name of the Provider). By this, Provider and Consumer have been connected: If you now view the properties of the Provider "Cars", you will see that the property "dependencies" has automatically been set to "CarReservation_entity" (you cannot edit it on this side).

Finally, we need to specifiy that the new Consumer Cars is to be used for the EntityField CAR_ID (in order to get a list of all cars in the edit/create mode): Open CarReservation_entity in the Navigator window, click on the EntityField CAR_ID and edit its property "consumer" by selecting "Cars". Done!

Dependencies between Entities are illustrated in an Entity-relationship diagram in the Editor window (upper middle part of the Designer, when you double-click on an Entity). You can rearrange, extend ("All dependencies"), save and print an Entity-relationship diagram via the respective buttons at the top of the Editor window.

Now, repeat the previous steps in order to create a dependency between CarDriver_entity and CarReservation_entity. Consistently, name both Provider and Consumer "CarDrivers".

That’s all: If you now open the CarReservation Context (menu item "car reservation") and click on the "plus" button, the EditView opens in edit mode, and you can make a selection both from all car datasets as well as from all driver datasets. The respective combo boxes' entries are generated according to the Entities' property "contentTitleProcess". (If the combo boxes shows empty lines instead, you have forgotten to set the contentTitleProcess of the providing Entity (see chapter "Configuring Entities" further above).

This is the simplest form of a dependency created by the Consumer-Provider mechanism. You will have noticed that no Parameter is involved here. This is because we need *all* car datasets and *all* driver datasets, i.e., we do not need a Parameter as criteria for selecting only a part of the datasets.
You will find more complete examples of dependencies in later chapters, especially in chapter "Complex dependencies".

Remember that you must always first deploy in order to make changes visible in the client.

10.1.2. Example: Car drivers and Persons

In the xRM project, a number of example persons are organised in the Entity "Person_entity" and displayed in the various Views of Context "Person". In terms of data modelling, it would mean data redundancy, if we created car driver datasets independently from person datasets: Features like name or date of birth already exist in a person dataset. We do not need to enter them again in driver datasets. Instead, we connect Driver_entity with Person_entity. To achieve this, we again use the Provider/Consumer mechanism. However, in this case, we can simply use the Person_entity’s existing Provider "Contacts":

Create a new Consumer for CarDriver_entity and name it "Persons". Set its Property "entityName" to "Person_entity", and "fieldName" to "Contacts" (this is a predefined Provider of the xRM project).

Then, set property "consumer" of CarDriver_entity’s field CONTACT_ID to "Persons".

> In most cases, it makes sense to give Provider and Consumer the same name. However, technically, this is not obligatory. You are free to name both Provider and Consumer according to your preferences.

After deploying, you will see in the client that in Context Car driver’s create dialog (click on blue "plus" button) a lookup table has automatically been added for field "Person". The reason for this automatism is that there is a View named PersonLookup_view, which is referenced in Context "Person"'s property "lookupview". This LookupView will automatically be used, whenever Person’s Provider "Contacts" is referenced in the Consumer configuration of another Context.

> As LookupViews often contain a high number of datasets, it is recommended to make sure that its content can be filtered. In the above example (PersonLookup_view) you can, e.g., enter the text "Smith" and press Enter, in order to see only datasets of persons named "Smith". This filter functionality is set in the RecordContainer of the Entity acting as Provider, by setting the respective RecordFieldMapping’s property "isLookupFilter" to true. (Note that this has only an effect, if property "isFilterable" is also set to true; the functionality is not available for EntityFields of contentType DATE). In the RecordContainer of Person_entity, the RecordFieldMappings "FIRSTNAME.value" and "LASTNAME.value" have properties "isFilterable" and "isLookupFilter" both set to true; therefore, you can filter persons in PersonLookup_view according to their first name as well as according to their last name.

In the LookupView, you can select - in cleartext - the person who acts as driver. Although the value of the related column CONTACT.CONTACTID is invisible, it is stored in field CONTACT_ID, as soon as you select a person and press "Save".

> If an EntityField gets its value via a Consumer, the LookupView of the Entity acting as Provider will always be used in the EditView, for selecting a value of this EntityField. If the providing Entity has no LookupView, then its contentTitleProcess will be used instead. And if there is also no contentTitleProcess set, then empty lines are shown in the combo box.

10.1.3. Retrieving pending records

(This is a little excursus, helpful to know in the context of dependencies, but not related to the carpool example project.)

10.1.3.1. Basics

"Pending records" are datasets ("records", "rows") that the user has entered in a View, but which have not been saved in the database yet. You can access them via the variables

- `insertedRows`
- `changedRows`
- `deletedRows`

to be read via, e.g., `vars.get("$field.MyConsumerName.insertedRows")`

Each of these variables returns a (often large) array of objects, with the Consumer field’s name as property. Here is an example of `changedRows` in Context OfferItem:

In most cases, you do not need the complete variable content, therefore here is an example of reading a consumed field’s value (in the context of the "Attribute" logic)

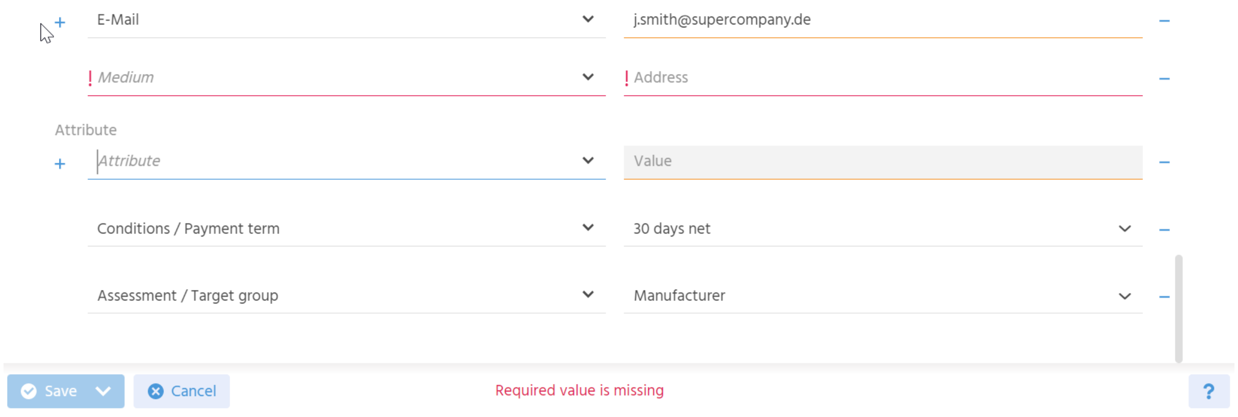
```
var changedRows = vars.get("$field.MyConsumerField.changedRows");
var myFieldValue = changedRows[0]["AB_ATTRIBUTE_ID"];
```

The field’s value is always returned as String. If a field is not set, you will get an empty String. Thus, a check for null or undefined is not required. But, if required, you still need to check for an empty string, e.g.

- via `if (myFieldValue === "") { … }` or
- via "TRUEish test", i.e., using JavaScript’s implicit type conversion:
  `if (myFieldValue) { … }`

10.1.3.2. Example 1

In the xRM project, if you use OrganisationEdit_view to enter or edit a company dataset, it is possible to assign multiple Attributes (e.g., target group, delivery terms) and communication channels (e.g., email, phone, website) to this company. These links are established via consumers.



Here, you can use the above variables in order to check

- if all mandatory Attributes are set
- if specific attributes have been assigned not more than once
- if a mandatory Attribute has been deleted
- if a change of an Attribute has violated a min/max rule
- etc.

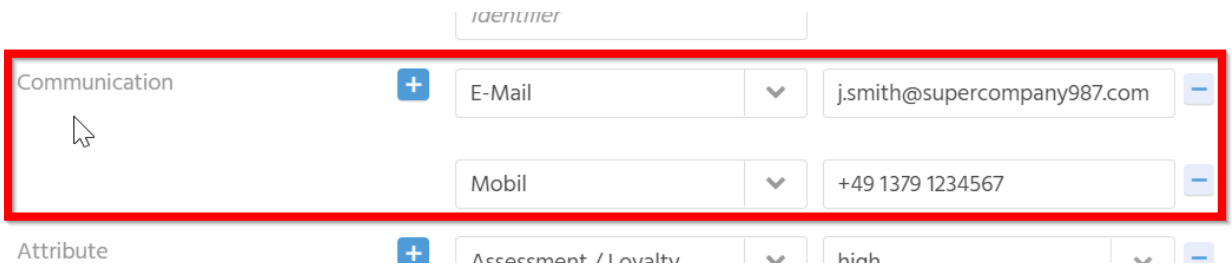
Then, the above variables are useful:

- `insertedRows`: These are the rows (here: attribute/value pairs) the user has entered, but not saved yet. As soon as they are saved, the rows will be removed from the variable.
- `changedRows`: These are rows (here: attribute/value pairs) that have already been saved in the database earlier, and now the user has changed (edited) them, but not saved the changes. As soon as the changes are saved, the rows will be removed from the variable. If the user enters a new row, without saving, and then changes it, the row will remain in variable `insertedRows`, and not be transferred to `changedRows`.
- `deletedRows`: These are the rows (here: attribute/value pairs) that are already in the database and that now have been marked by the user as "to be deleted" (e.g., by clicking the "minus" icon to the right of the attribute/value pair). As soon as the user clicks the save button, the row will be removed from the variable.
  If the user enters a new row, without saving, and then deletes it, the row will not appear in any of the 3 variables.

10.1.3.3. Example 2

Here is another example, which is also included in the xRM project and is similar to the above example:

In PersonEditView, the client user can insert multiple communication channels related to a contact person (email address, mobile phone number, etc.):



Technically, this is realized via Person_entity's Consumer "Communications", which relates to Provider "AllCommunications" of "Communication_entity".

If, e.g., the user has entered the 2 "Communication" datasets as shown above, but has not yet pressed the "Save" button, you can nevertheless already retrieve them via variable `insertedRows`. In the xRM project, this is done, e.g., in the context of the logic that finds duplicates.

Example of variable `insertedRows` *in Person_entity.DuplicatesPerson.DuplicateObject_param.valueProcess.js*

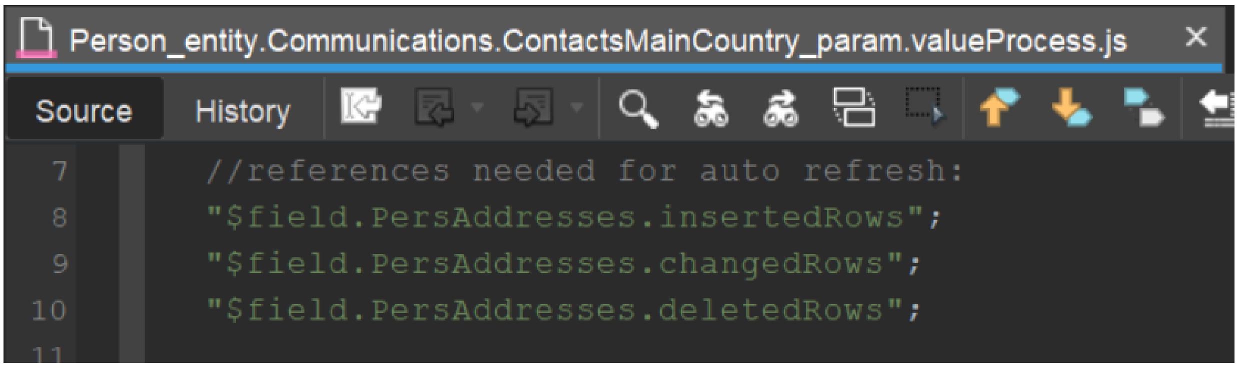
```
var communications = vars.get("$field.Communications.insertedRows");
```

10.1.3.3.1. EntityConsumerRowsHelper

In Entity_lib (see process > libraries) you can find the "EntityConsumerRowsHelper", which simplifies the usage of the above 3 variables. For example, by calling `EntityConsumerRowsHelper.getCurrentConsumerRows` you can load *all* datasets that are currently visible to the user (including all possible changes that the user might have done, with or without saving them). Instead, when reading the 3 variables directly via `vars.get(…)` you will only get the *changed* datasets, not *all* of them.

10.1.3.3.2. Implicit refreshing

Calling the above variables is also used for marking a specific process as being *implicitly* dependent of one or all of these variables and thus establishing an efficient auto-refresh. Here is an example included in the xRM project:


Person_entity.Communications.ContactsMainCountry_param.valueProcess.js
Source
History

```
//references needed for auto refresh:
"$field.PersAddresses.insertedRows";
"$field.PersAddresses.changedRows";
"$field.PersAddresses.deletedRows";
```

The advantages of these implicit dependencies are:

- Avoiding an explicit refresh, which would *always* trigger a re-calculation, even if this is not necessary in every case (which, in turn, *decreases the system's performance* - see AID066 Performance Optimization).
- Restricting refreshes to cases that *actually* include changes.

You can think of this kind of refresh definition as being similar to the "meta information" commonly used in various frameworks (in Java mostly defined via annotations).

10.1.3.4. Further information

Find more information on these variables via

- their JSDoc: Type, e.g., `EntityConsumerRowsHelper.` or `vars.get("$field.MyConsumerField.` and press CTRL+SPACE, then you will see the available methods/variables; and if you select one of them, you their documentation will be displayed as JSDoc.
- performing a full-text search over the complete xRM project, using the methods'/variables' names as search term: You will find various implementation examples that will help you to understand the functionality even better.

10.2. Using keywords (predefined values)

Now we proceed to another task, which can be performed by using the Provider-Consumer principle, and this time also a Parameter is required:

In some cases it is useful to restrict the field values that can be entered to a limited number of predefined values. E.g., we want to avoid that the same manufacturer is spelled in different ways for different cars, e.g. "Mercedes" and "Daimler Benz", or, as for colors, "Red" and "red". This would lead to inconsistencies and disturb data filtering.

In ADITO, sets of predefined values are called "keywords". A single keyword consists of

- one **"keyword category"**, see KeywordCategory_entity and the corresponding database table AB_KEYWORD_CATEGORY. In the client, you can create a keyword category via Administration > Keyword Category. In our example, the category is "CarColor". Each category is identified by its AB_KEYWORD_CATEGORY_ID, which is a UID. It has further fields in order to define, e.g.,
  - the sorting: SORTINGBY (0 = "manual", see below; 1 = by title; 2 = by translated title)
- one or multiple **"keyword entries"**, see KeywordEntry_entity and the corresponding database table AB_KEYWORD_ENTRY. In the client, you can create a keyword entry via Administration > Keyword Entry. Keyword entries are the selectable values of a keyword, e.g., "green", "red", and "blue". Each keyword entry has
  - a unique identifier AB_KEYWORD_ENTRYID, which holds a UID
  - an additional, non-unique identifier KEYID, whose value is cleartext, e.g. "GREEN"
  - a TITLE, which is used in the selection components (combo boxes, etc.) of the client, e.g. "green"
  - further fields for defining, e.g., the position in a selection list (when "manual" sorting is set in the keyword category), or the system relevance ([ISESSENTIAL]), meaning whether or not a keyword entry is used in the code.
- one or multiple **"keyword attributes"**, see KeywordAttribute_entity and KeywordAttributeRelation_entity and the corresponding database tables AB_KEYWORD_ATTRIBUTE and AB_KEYWORD_ATTRIBUTERELATION. In the client, you can create a keyword attribute via Administration > Keyword Attribute. Each keyword attribute is created for one specific keyword category and has a specific data type (e.g., Boolean). The effect is, that for every keyword entry of this keyword category the client administrator can then assign this attribute and set an attribute value.
  We will not use keyword attributes for our carpool example, but you may look at an example in the xRM project: COMMOBIL is one of several keyword entries of category CommunicationMedium. "contentType" is the name of one of several keyword attributes defined for the category CommunicationMedium; "contentType" has the data type "String". This allows the client administrator to assign this keyword attribute "contentType" to keyword entry COMMOBIL and set a free text value for it, namely "TELEPHONE". This in turn is then used for postprocessing logic, e.g., for formatting purposes (see, e.g., valueProcess of Communication_entity’s EntityField ADDR).
  NOTE: Do not mix up datasets of "keyword attributes" (KeywordAttribute_entity) with datasets of "attributes" (Attribute_entity). These are 2 completely separate parts of the xRM project, and they have completely different purpose and handling.

Further on, you will learn how to make the values (entries) of a keyword selectable for a specific

EntityField. In the EntityField, only a reference to the keyword entry (the so-called KEYID) is stored, but an automatism makes sure that, in the client, the keyword entry’s TITLE is displayed.

Thus, the first step is to insert some new datasets in the database tables AB_KEYWORD_CATEGORY and AB_KEYWORD_ENTRY. Again, we use Liquibase. Let’s start with the colors.

Any keyword that is used in the code (process, library, etc.) of an ADITO project, must be marked as "essential", by setting the corresponding database field AB_KEYWORD_ENTRY.ISESSENTIAL to value "1", which correspondents to "Yes" ("true"). This makes sure that even the client administrator cannot delete the keyword in the web client ("Delete" button/option is not active/shown in the Views of Context KeywordEntry), because otherwise errors could occur.

If you want to add flexible features to the datasets of an Entity (e.g., like in this case, the color of a car, along with selectable values "green", "red", "blue", etc.) you have, in principle, at least the following 2 options:

- Add an additional EntityField, which is related to Keywords Entries (like we will do it in our carpool example).
- Make the client-side setting of **Attributes** available.

Each approach shows notable advantages and disadvantages, in particular, as performance and usability are concerned. Therefore, we strongly recommend you to read appendix "EntityField/Keywords vs. Attributes", after you are finished with the carpool example project..

10.2.1. Example: Car colors

In the "Projects" window, open the folder alias > Data_alias > "example_carpool". In this folder create a new changeset XML file and name it "init_carcolor" (the extension "xml" will remain/be added automatically).Replace the default code by the following code:

init_carcolor.xml

```
<?xml version="1.1" encoding="UTF-8" standalone="no"?>
<databaseChangeLog xmlns="http://www.liquibase.org/xml/ns/dbchangelog" xmlns:ext="http://www.liquibase.org/xml/ns/dbchangelog-ext"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog-ext
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-ext.xsd http://www.liquibase.org/xml/ns/dbchangelog
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.6.xsd">
    <changeSet author="j.smith" id="aaa90096-2a99-4e64-9a5c-9d0a18211cf3">
        <insert tableName="AB_KEYWORD_CATEGORY">
          <column name="AB_KEYWORD_CATEGORYID" value="2aa33f21-6aad-4bea-9106-7265db39e052"/>
          <column name="NAME" value="CarColor"/>
          <column name="SORTINGBY" valueNumeric="0"/>
          <column name="SORTINGDIRECTION" value="ASC"/>
        </insert>
    </changeSet>
    <changeSet author="j.smith" id="075f14fa-67a6-4841-8d6b-c85b7b576e82">
        <insert tableName="AB_KEYWORD_ENTRY">
```

```
            <column name="AB_KEYWORD_ENTRYID" value="2324a950-6767-4366-abf5-a343b7fd11f3"/>
            <column name="KEYID" value="RED"/>
            <column name="TITLE" value="red"/>
            <column name="AB_KEYWORD_CATEGORY_ID" value="2aa33f21-6aad-4bea-9106-7265db39e052"/>
            <column name="SORTING" valueNumeric="0"/>
            <column name="ISACTIVE" valueNumeric="1"/>
            <column name="ISESSENTIAL" valueNumeric="1"/>
        </insert>
        <insert tableName="AB_KEYWORD_ENTRY">
            <column name="AB_KEYWORD_ENTRYID" value="0c89fdc7-63cb-4e2b-a65a-ad77d3e58cc2"/>
            <column name="KEYID" value="YELLOW"/>
            <column name="TITLE" value="yellow"/>
            <column name="AB_KEYWORD_CATEGORY_ID" value="2aa33f21-6aad-4bea-9106-7265db39e052"/>
            <column name="SORTING" valueNumeric="1"/>
            <column name="ISACTIVE" valueNumeric="1"/>
            <column name="ISESSENTIAL" valueNumeric="1"/>
        </insert>
        <insert tableName="AB_KEYWORD_ENTRY">
            <column name="AB_KEYWORD_ENTRYID" value="0544cce6-170b-43c2-b9bc-17c89a42c15f"/>
            <column name="KEYID" value="GREEN"/>
            <column name="TITLE" value="green"/>
            <column name="AB_KEYWORD_CATEGORY_ID" value="2aa33f21-6aad-4bea-9106-7265db39e052"/>
            <column name="SORTING" valueNumeric="2"/>
            <column name="ISACTIVE" valueNumeric="1"/>
            <column name="ISESSENTIAL" valueNumeric="1"/>
        </insert>
    </changeSet>
</databaseChangeLog>
```

(Remove possible line breaks after copying the code.)

Explanations:
At first, a keyword category is created and named "CarColor". Afterwards, the keyword entries are created: Column AB_KEYWORD_ENTRYID holds the primary key (mostly a UID), while the values of column KEYID are not unique, i.e., they can (in principle) be used for multiple AB_KEYWORD_ENTRY datasets. The column AB_KEYWORD_CATEGORY_ID holds the UID of the category of the keyword; therefore, its value is the same for all keyword entries referring to the same keyword, here, "CarColor". TITLE is the cleartext name of a single keyword entry, identified by KEYID. As we might use these keywords in our ADITO project’s code (processes, etc.), we set AB_KEYWORD_ENTRY.ISESSENTIAL to value "1", which correspondents to "Yes" ("true") - see note above. The sorting is set to "manual" (meaning "arbitrarily defined for each keyword entry):

- in the keyword category dataset, SORTINGBY gets value 0 (= "manual", meaning that the value of the keyword entry’s field SORTING is to be respected (see below)
- in the keyword entry datasets, the values of field SORTING are set arbitrarily: "0" for RED, "1" for YELLOW, and "2" for GREEN.
  This will have the effect that, in the client, the corresponding selection list of TITLEs will be shown in the defined order
  red
  yellow
  green
  NOTE: "Manual" sorting also means that this order will not change even in case the titles are submitted to automatic translation (see below). (If you prefer the latter option, you need to set SORTINGBY to value 2, which means "select by translated TITLE").

As mentioned before, usually, the KEYID is saved in the database, while specific configurations and processes (see below) make sure that, in the client, the TITLE is displayed instead of the KEYID. And it’s the TITLE that can be submitted to translation logic (see language files in the project’s folder "language").

In order to include the above xml file in the Liquibase logic, proceed as usual: In the "Projects" window, in the "example_carpool" (!) folder (alias > Data_alias > example_carpool), extend the code of changelog.xml by the following line:

```
(...)
<include relativeToChangelogFile="true" file="init_carcolor.xml"/>
```

Remember that there are multiple XML files named "changelog.xml". Do not mistake them.

Perform a Liquibase update via the context menu of "Data_alias". If required, clear the server cache in order to see the new entries.

In principle we could, in our project’s code, relate to the new keyword in cleartext ("CarColor"). However, for multiple reasons (e.g., consistency in the case of renaming), it is better to define all keywords' names in a central location. For ADITO xRM, this is the library KeywordRegistry_basic (in the "Projects" window, under process > libraries).

System-wide processes like KeywordRegistry_basic can, in principle, be customized according to the project’s requirements. However, this can lead to update/merge problems whenever ADITO releases a new xRM version. **Therefore, we strongly recommend to leave all ADITO xRM processes/libraries unchanged** and create new processes for any customized functionality, using project-specific prefixes for the naming.

Therefore, for handling our carpool keywords, we create a new library named KeywordRegistry_carPool (process > new ...; set process property "variants" to "libraries"). Then, we open the new library KeywordRegistry_carPool and insert the following code:

KeywordRegistry_carPool

```
/* Keywords for car pool example */

export function $KeywordRegistryCarPool(){}

// Keyword category
```

```
$KeywordRegistryCarPool.carColor = function(){return "CarColor";};
```

This logic works similar to what you might know as "Enumeration" or "Constant": Instead of using the keyword’s cleartext directly, we can now call the function $KeywordRegistryCarPool.carColor which returns the keyword’s cleartext.

Beside a keyword category, you can also specify code to retrieve the entries of this keyword (KEYID), i.e., in our case, the names of the colors. The syntax is as follows:

KeywordRegistry_carPool

```
// Keyword entry names (KEYID)
$KeywordRegistryCarPool.carColor$red = function(){return "RED";};
$KeywordRegistryCarPool.carColor$yellow = function(){return "YELLOW";};
$KeywordRegistryCarPool.carColor$green = function(){return "GREEN";};
```

However, you should reference a KEYID here only if it is actually used in the code. To prevent exceptions (e.g., after deletion by mistake), these kind of KEYIDs must also be stored with column ISESSENTIAL set to value 1 (= "true"), which has been already done for the colors (see above Liquibase file), because we will need it for an example in chapter "Color", subchapter of Controlling the design.

Under process > libraries you can find libraries including helper methods. If the name is simply MyContextName_lib, e.g. Organisation_lib, the purpose of the methods is mainly restricted to the Context. However, if the name includes "Util", then the library includes helper methods that are of general use, e.g. FileUtil_lib or Util_lib. Most of the methods included in these libraries have a JSDoc explaining details of how to use them. Furthermore, also the ADITO platform ("core") provides helper methods - see the overview given in the documentation that you can access via menu Help > Show Documentation.

Now that this preparatory work is done, we can go on with connecting the COLOR field with the corresponding keyword entries, using the Consumer-Provider principle, along with a specific Parameter:

First, look at the existing configuration of the Entity KeywordEntry_entity: Double-click on it in the "Projects" window and look at the Navigator window: You see that there already exists a Provider named "SpecificContainerKeywords", which shows the following properties:

- dependencies: a readonly property showing all Entities that have a Consumer related to this Provider (see Consumer’s property "fieldName")

- lookupIdField: KEYID (= the field containing the keyword entries to be shown)
- recordContainer: db and jDito (= the RecordContainers of KeywordEntry_entity)

Furthermore, you will recognize that the Provider "SpecificContainerKeywords" has a Parameter named "ContainerName_param" assigned. This Parameter has its property "expose" set to true.

> You can create a new Parameter in the Navigator window, if you right-click on an Entity and choose option "New Parameter" in the context menu. After you have named it, it appears in the folder "Parameters". If you set its property "expose" to true, the Parameter will also appear (in grey font color) under every Provider of the same Entity, as well as under every Consumer of other Entities, if the Consumer references the Provider. The grey font color means that the Parameter is not yet initialized; to set its parameters (especially, its valueProcess), right-click on the grey-fonted Parameter and choose "Initialize" from the context menu. (If you want to undo the initialization, right-click on the Parameter again and then choose option "Restore Default Value". This will reset its font color from white to grey again, indicating that it is not initialized.)

We will use this existing Parameter in the Consumer logic of Car_entity:

Create a new Consumer for Car_entity and name it "KeywordCarColors". Edit the new Consumer's properties as follows:

- EntityName: select "KeywordEntry_entity"
- fieldName: select "SpecificContainerKeywords"

Specify this new Consumer "KeywordCarColors" in the "consumer" property of Car_entity's field "COLOR".

Automatically, in the Navigator window, the above mentioned Parameter "ContainerName_param" will appear under the Consumer "KeywordCarColors". In the Parameter's property "valueProcess", we enter the following code:

ContainerName_param.valueProcess

```
result.string($KeywordRegistryCarPool.carColor());
```

These are the steps to create a relation (dependency) between two Entities, here, between KeywordEntry_entity and Car_entity.

If we test our changes in the client (don't forget to deploy first), we see that in the create dialog (click on blue "plus" button), the field "Color" shows a combo box, in which you can select the colors in clear text. However, if you create a new car dataset (including the selection of a color) and save it, all Views do not display the color as cleartext, but instead they display the KEYID (here: a UID) corresponding to the color. This will be explained later.

10.2.2. Example: Manufacturers

Now, repeat the steps of the previous chapter in order to turn the field MANUFACTURER from a free text field into a field restricted to a number of predefined manufacturers' names. Again, you can use Liquibase to create the required keyword entries - which remains the preferred way, as in the case of, e.g., a database change, you can easily re-create the keyword entries.

Alternatively - e.g., for testing purposes - you can also generate new keyword entries via the client:

- In the "Administration" group of the Global Menu, choose Context "Keyword Category". Click on the "Create new" button (blue "plus") and look to the right: Instead of the PreviewView, a so-called "SmallEditView" appears (Note: In this case, the EditView does not pop up as separate window, but it appears directly to the right of the FilterView; these options are controlled by the View property "size", which is by default set to NORMAL, but in this case is set to SMALL). Now, create a new KeywordCategory and name it, e.g., "CarManufacturer".
- Then, again in the "Administration" group of the Global Menu and choose Context "Keyword Entry".
    - Click on the "Create new" button (blue "plus") and select the Keyword Category that you have just created in the previous step (e.g., "CarManufacturer")** Then, create a first Keyword (= "Keyword Entry"):
        - Keyword Category: the KeywordCategory that you have just created (e.g., "CarManufacturer"):
        - Title: The title of the Keyword to be used for selecting it in the client. To be more precise, this is the respective "Key" in the language files (see chapter Language files), enabling the Keyword to appear in the respective login language. It will be saved in the database column AB_KEYWORD_CATEGORY.NAME.
          Example: "Mercedes".
        - Key: The identificator of the Keyword, to be used in code/processes to be configured via the Designer. It will be saved in the database column AB_KEYWORD_ENTRY.KEYID.
          Example: "MERCEDES".
        - Leave the flag "Active" to true.
    - Create further Keywords for the same KeywordCategory, e.g., "BMW", "FORD", etc.

Now that you have integrated keyword logic for Car_entity fields MANUFACTURER and COLOR, you should also adapt property contentTitleProcess of Car_entity (otherwise, UIDs are displayed instead of the names of the manufacturers/colors in cleartext):

Car_entity.contentTitleProcess

```
import { result, vars } from "@aditosoftware/jdito-types";
import { $KeywordRegistryCarPool } from "KeywordRegistry_carPool";
import { KeywordUtils } from "KeywordUtils_lib";


var carManufacturer = KeywordUtils.getViewValue($KeywordRegistryCarPool.carManufacturer(), vars.get(
"$field.MANUFACTURER"));
var type = vars.get("$field.TYPE");
var color = KeywordUtils.getViewValue($KeywordRegistryCarPool.carColor(), vars.get("$field.COLOR"));
result.string(carManufacturer + " " + type + " " + color);
```

10.2.3. Example: Currency

To implement predefined currency names is even easier, because the ADITO xRM project already includes a keyword named "Currency", with several entries, e.g., "Euro". Furthermore, the library "KeywordRegistry_basic", already includes a function returning "Currency", and KeywordEntry_entity includes a Provider named "SpecificContainerKeywords", which you can reference as Consumer.

You can easily extend the list of available currencies in the client, via Context "Keyword Entry" (see menu group "Administration").

Now, try to make currencies selectable both in Context Car and in Context CarReservation.

10.3. Controlling the displayed value

In some cases, we do not want to see exactly the database value referring to the EntityField, but a value derived or calculated from it. As for keywords, e.g., instead of the KEYID, the TITLE of the keyword entry is to be displayed. Likewise, the user wants to read the drivers' names rather than their CARDRIVERID or CONTACT_ID.

This transformation of an ID value into a displayed value can, in principle, be done at 2 different locations:

- in the <field name>.displayValue field of a RecordContainer
- in the displayValueProcess of an EntityField

10.3.1. displayValue of a RecordContainer field

For every EntityField, 2 automatically prepared so-called RecordFieldMappings exist in the

RecordContainer (sometimes simply called "RecordFields"):

- <field name>.value: In property "recordfield", you specify the database field to be used for saving (persisting) the corresponding value. We have already applied this in previous chapters.
- <field name>.displayValue: In property "recordfield", you specify the database field whose value is to be displayed. If you leave it empty, property "recordfield" of the "…value" field is also used for displaying purposes. **NOTE: The displayValue is not formatted automatically (unlike the value).**

Both fields have a process property named "**expression**": Here, you can enter any SQL code valid in the "select" part of the SQL statement - for example a fix value, a fully-qualified database column name, or (a common use case) an SQL subselect that returns the preferred value/displayValue. Technically, the result of the "expression" process is simply inserted in the SQL's "select" part - with a leading comma, an opening parenthesis and a closing parenthesis automatically added (so you do not need to include outer parantheses again in the "expression").

If you have activated database logging (see chapter Logging), you can inspect the SQL generated by ADITO on the basis of the configuration of the RecordContainer, e.g. something like this:

Generic example of SQL (DB logging output) when property "expression" has been configured

```
SELECT MYTABLE1.MYCOLUMN1 , MYTABLE1.MYCOLUMN2 , MYTABLE1.MYCOLUMN3 , ( select MYTABLE2.MYCOLUMN4 from MYTABLE2 where MYTABLE2
.MYTABLE2ID = MYTABLE1.MYTABLE2_ID )
FROM MYTABLE1
ORDER BY <...>     LIMIT 400
```

In many cases, it is not necessary to write these subselects by yourself, but you can use existing helper functions that return SQL code (see, e.g., the functions included in project folder process > libraries).

In a RecordFieldMapping, properties "recordField" and "expression" should never be set both at the same time, because "recordField" will then always be preferred by the ADITO logic, meaning that the code entered in "expression" will always be ignored in this case.

Be aware that using the displayValueProcess of an EntityField can be a **performance killer**:
Whenever you want to control the display of a feature, using the displayValue's "expression" property usually enables a **higher performance** than other ways, in particular, than using property displayValueProcess of the EntityField (see below): The technical background is that the former simply extends the SQL statement of the loading process, while the latter initiates an additional loading process, which is executed separately for every single dataset.

However, if you want to create a new dataset that includes a predefined value for a specific field (or a combo box for selecting from a list of values), then setting "expression" is not enough, but you additionally need to set the displayValueProcess for this field.
To sum it up: Using the displayValueProcess of an EntityField can be a performance killer. Thus, as a rule of thumb, you should **always** try to retrieve the display value via SQL in the RecordContainer, using the "expression" property. If you need a display value at creation time, you can additionally (not alternatively!) program a displayValueProcess. If both "expression" and displayValueProcess are filled, the ADITO system automatically ignores the displayValueProcess (but not the valueProcess!), whenever a display value can be retrieved via "expression".
Find related information in appendix "Accessing the value of an EntityField".

A displayValue cannot be deactivated at runtime. If, for one and the same EntityField, in some ViewTemplates the displayValue is to be shown and in other ViewTemplates only the value (not the displayValue), then you can realize this by simply creating 2 separate EntityFields that load the same value, with a displayValue only being set for one of them.

10.3.1.1. Example: Driver’s name

CarDriver_entity.db.CONTACT_ID.displayValue.expression

```
result.string(PersUtils.getResolvingDisplaySubSql("CONTACT_ID"));
```

The effect of this code is that not the CONTACT_ID is displayed, but the full name and salutation of the corresponding person. This is achieved via a helper function, included in the library Person_lib (under process > libraries). If you are interested to know what SQL code this helper function returns, please refer to appendix "Database Access", chapter "SQL Helper Functions".

Now that you have added this displayValue, you can refer to it and thus simplify the code of other processes, e.g., of the Entity’s contentTitleProcess:

CarDriver_entity.contentTitleProcess

```
result.string(vars.get("$field.CONTACT_ID.displayValue"));
```

10.3.1.2. Example: Manufacturer

Car_entity.db.MANUFACTURER.displayValue.expression

```
var sql = KeywordUtils.getResolvedTitleSqlPart($KeywordRegistryCarPool.carManufacturer(), "CAR.MANUFACTURER");
result.string(sql);
```

The effect of this code is that, as for the keyword "Manufacturer", not the KEYID is displayed, but the TITLE (cleartext, e.g., "Mercedes"). The helper function `getResolvedTitleSqlPart` returns the required subselect for a given keyword ("Manufacturer", returned by `$KeywordRegistryCarPool.carManufacturer()`) and a given database column ("CAR.MANUFACTURER"). (If you are interested to know what SQL code this helper function returns, please refer to appendix "Database Access", chapter "SQL Helper Functions".)
Before, you must add the line

```
$KeywordRegistryCarPool.carManufacturer = function(){return "Manufacturer";};
```

at the end of the existing code of library KeywordRegistry_carPool (under process > libraries).

10.3.1.3. Example: Car color

Now, try the same for car colors, on your own. Again, we do not want to see the KEYID, but the TITLE (cleartext, e.g., "red").

Here is the solution:

Car_entity.db.COLOR.displayValue.expression

```
var sql = KeywordUtils.getResolvedTitleSqlPart($KeywordRegistryCarPool.carColor(), "CAR.COLOR");
result.string(sql);
```

The effect of this code is that, as for the keyword "CarColor", not the KEYID is displayed, but the TITLE (cleartext). Function `getResolvedTitleSqlPart` returns the required subselect for a given keyword ("CarColor", returned by `$KeywordRegistryCarPool.carColor()`) and a given database column ("CAR.COLOR").

Please note that the color might not be visible in cleartext in the EditView, unless you have entered a displayValueProcess of the EntityField COLOR. This will be explained in a later chapter.

10.3.1.4. Example: Currency

If you view the entries of keyword "Currency" (with the database editor or in the client, choosing

Administration > Keyword Entry), you will recognize that the KEYID is not a UID, but the (also unique) ISO 4217 Currency Codes, e.g. "EUR" or "USD". This makes it easier, as we want to use these codes for display purposes, rather than the long versions "Euro" or "United States dollar". However, the latter is automatically displayed when creating or editing a dataset; this is due to an automatism on the Provider Entity for keywords (see KeywordEntry_entity.contentTitleProcess), because in most cases TITLE is to be displayed instead of KEYID.

10.3.2. displayValueProcess of an EntityField

Another way to control the display of a field is the EntityField’s property "displayValueProcess".

> If the displayValue field of a RecordContainer (e.g., "expression") is set, then the displayValueProcess of the EntityField is automatically ignored in most cases. It is only executed, if a display via the RecordContainer (which mostly shows the higher performance) is not possible. The latter, e.g., happens when creating an new dataset and, in the EditView, a specific value is to be shown (e.g., the car color in cleartext instead of its KEYID) or preselected. This is because the displayValue field of a RecordContainer is related to *existing* datasets, which means, it has no effect when entering a *new* dataset (which does not exist in the database unless the "Save" button is clicked); therefore, in this case, you need the displayValueProcess of the EntityField.
> There might be improvements in later ADITO releases, but currently it is recommended always to fill in both the "displayValue.expression" process of the RecordContainer and the displayValueProcess of the EntityField, whenever you want to control the display of a field.

10.3.2.1. Example: Car Color

To view, e.g., the color in cleartext, enter the following code in the property displayValueProcess of Car_entity’s field COLOR:

Car_entity.COLOR.displayValueProcess

```
result.string(KeywordUtils.getViewValue($KeywordRegistryCarPool.carColor(), vars.get("$field.COLOR")));
```

Explanations:

- `$field.COLOR` specifies the field, whose value (here: the KEYID) is to be "translated"
- `$KeywordRegistry.carColor()` returns the category of the keyword, whose keyword entries hold the "translation" (i.e., KEYID and TITLE)

* `KeywordUtils.getViewValue` returns the "translated" value (TITLE) of the given KEYID

> Now that you know how to "translate" a keyword entry, you can optimize other parts of the client, e.g., add a similar displayValueProcess for Car_entity's field MANUFACTURER.

10.3.2.2. Example: Currency

Here is an example how to display the price of the car along with the currency, in one single field (PRICE).

Car_entity.PRICE.displayValueProcess

```
if (vars.get("$this.value") !== null)  {

  var myPrice = vars.get("$this.value");
  result.string(text.formatDouble(myPrice, "#,##0.00") + " " + vars.get("$field.CURRENCY"))
}
```

Explanations:

* `vars.get("$this.value")` returns the current value of the corresponding EntityField (here: field PRICE). Find more information in appendix "Accessing the value of an EntityField".
* `if (vars.exists("$this.value"))` is required to avoid exceptions in case a value of PRICE does not yet exist (e.g., when creating a new car dataset)
* `text.formatDouble(myPrice, ",#0.00")` formats the car's price according to the given pattern. The pattern is always to be specified the English way, i.e., a comma as thousands separator, and a point as decimal separator. This format will be adapted automatically, depending on your browser's language settings.

10.4. Complex dependencies

ADITO enables you to combine almost arbitrary Views and establish arbitrary dependencies between them. And, in principle, the dependencies can be arbitrarily nested - i.e., you can define that a View depends on another View, which in turn depends on a third View etc. Using these combinations, powerful applications can be built. We will give you a few examples.

10.4.1. MasterDetailLayout

The MasterDetailLayout enables you to specify one View as "master", on which one or more other Views ("details") depend.

For example, if you open the Context "Company" (in the client, in menu group "Contact Management"), you select a company dataset and then press the "Open" button. This will open the Context’s MainView, which has a MasterDetailLayout: On the left, you see the "master", which is the PreviewView of the company. On the right, you see the "details", which are several Views, sorted in tabs, e.g., activities, contacts, or attributes. These "detail" Views belong to other Contexts, e.g., Activity, Person, or AttributeRelation.

Thus, a View having a "MasterDetailLayout" often has no own View elements, but is used as a kind of frame connecting one View (with one dataset) with one or more dependent Views (each showing one or multiple datasets).

In most cases, an Entity’s MainView has a "MasterDetailLayout",

- with the PreviewView of the same Entity being the "master" and
- various Views of the same or of other Entities - often in table form - being the "details".

Note that DashletConfigs cannot be added to Views having a MasterDetailLayout.

Now, how to configure such a combination of dependent Views via a MasterDetailLayout?

At first (after creating Context and Entity), in the "Projects" window, we create a View (usually this will be the MainView) and set its property "layout" to "MasterDetailLayout". Consequently, a property named "master" will appear below. This will be set later.

Now we define both the "master" View and all "detail" Views:

Double-click on the View having the "MasterDetailLayout" in the "Projects" window. Then, in the Navigator window, right-click on the name of this View and choose "Add reference to existing View…". A dialog with the following lines will appear:

- EntityField: In this combo box there are one or more options to select:
  - "#ENTITY": Choose this option, if you want to add a View of the same Entity (i.e., of the Entity related to the View having the "MasterDetailLayout")
  - (if existing:) All Consumers of the Entity related to the View having the "MasterDetailLayout". Choose a Consumer if you want to add a View of another Entity, i.e., of the Entity related to the Consumer. In many cases you must first create this Consumer (and possibly also the related Provider), which is explained below.
- View: All Views related to the selection in the above line "entityField".
- Assign to: Leave empty.

Now, we edit the property "master" of the View having the MasterDetailLayout and set it to the View that should act as "master". All other referenced Views will automatically be treated as "detail" Views.

In the Navigator window, the "detail" View appears on the same level as the "master" View, however, without a prefix in the name.

In the client, the "master" View usually appears on the left, while the "detail" related Views appear on the right, sorted in tabs.

The dependency between the "master" View and a "detail" View is established using the Provider-Consumer mechanism: The Provider provides (delivers) "detail" data selected according to a Parameter specified by the Consumer, which here is the "master" View. (The Provider "exposes" a Parameter, whose value process is set on Consumer side and works as selection criteria of what data is to be provided). Technically, this works as follows:

- The **Provider side** is the Entity (!) of the "detail" View. Here,
  - a Parameter must be created, whose name usually refers to the selection criteria specified by the "master". If, e.g., the Provider provides data of contact persons working in a specific organisation, the Parameter could be named "OrgId_param". Furthermore, this Parameter must be exposed (property "expose" = true), i.e. "offered" to the Consumer.
  - a new Provider object must be created, whose name usually refers to the provided data. If, e.g., the Provider provides data of contact persons, it could be named "Contacts".
  - the conditionProcess of the RecordContainer must process the Parameter by including it as data selection criteria. Here is an example how this piece of code usually looks like:

    XXX_entity.RecordContainers.db.conditionProcess

    ```
    var cond = newWhereIfSet("CONTACT.ORGANISATION_ID", "$param.OrgId_param");
    ```

```
result.string(cond);
```

In short, this code means: "If Parameter OrgId_param exists (= is "filled"), then return only CONTACT datasets showing the ORGANISATION_ID handed over in the Parameter. If the Parameter OrgId_param does not exist (= is not "filled"), then no condition is built, so *all* CONTACT datasets are returned.

- The **Consumer side** is the Entity (!) of the "master" View. Here,
  - a Consumer object must be created, whose name is often identical with the name of the Provider. If, e.g., the Provider provides data of contact persons, the Consumer could also be named "Contacts". The new Consumer gets a reference to the Provider, by setting its properties "entityName" and "fieldName" (i.e., Provider name) accordingly, e.g. "Person_entity" and "Contacts". Consequently, all Parameters exposed by the specified Provider will be visible "under" the Consumer, in grey font color.

    The grey font color means that the Parameter is not yet initialized; to set its parameters (especially, its valueProcess), right-click on the grey fonted parameter and choose "Initialize" from the context menu. (If you want to undo the initialization, right-click on the Parameter again and then choose option "Restore Default Value". This will reset its font color from white to grey again, indicating that it is not initialized.)

  - set the respective Parameter's property "valueProcess" accordingly. E.g., the valueProcess' code for Parameter "OrgId_param" could look like this:

    XXX_entity.Consumers.XXX.XXXId_param.valueProcess

    ```
    result.string(vars.get("$field.ORGANISATIONID"));
    ```

    This will change the font color of the Parameter name from grey to white.

- The dependency configured on the Consumer side is now also visible in the property "dependencies" of the Provider object.

Now that the dependency has been established, all Views of the Consumer side are also available for selection in the dialog "Add reference to existing View…" of the View having the MasterDetailLayout. E.g., for OrganisationMain_view we could now add a reference to CarDriverFilter_view, by selecting the Consumer "Contacts" in the dialog's combo box "EntityField" and then selecting "CarDriverFilter_view" in the combo box "View" (leave field "Assign To" empty).

10.4.1.1. Example: Showing all reservations of a driver in the MainView

Let’s extend Context driver by a View showing all reservations of a specific driver in the CarDriverMain_view. As soon as the user opens a driver in the MainView (by selecting a driver and then pressing the "Open" button), all the driver’s reservations should be displayed.

To achieve this, we use the functionality of the "MasterDetailLayout". In an earlier chapter, we had already assigned this layout to all MainViews of our application. However, at that time we had only specified that the respective Entity’s PreviewView is the "master" of the layout. Now we come to the "detail" part of the MasterDetailLayout: We would like to specify CarReservationFilter_view as detail View, depending on CarDriverPreview_view ("master").

If you try to do this immediately, by choosing "Add reference to existing View…" in the context menu of CarDriverMain_view (Navigator window), you will notice that here we cannot select CarReservationFilter_view yet. To make it available for selection, we first need to establish a dependency between CarDriver_entity and CarReservation_entity, using the Provider-Consumer mechanism.

At first, we handle the Provider side:

In the "Projects" windows, double-click on CarReservation_entity, so it will be opened in the Navigator window. Here, right-click on CarReservation_entity and choose "New Parameter" from the context menu. Name the Parameter "CarDriverId_param" and set its property "expose" to "true".

Again, right-click on CarReservation_entity and choose "New Provider". Name the Provider "CarReservations".

Click on the RecordContainer "db" and edit its property "conditionProcess" by inserting the following code:

CarReservation_entity.RecordContainers.db.conditionProcess

```
import { result } from "@aditosoftware/jdito-types";
import { newWhereIfSet } from "SqlBuilder_lib";

var cond = newWhereIfSet("CARRESERVATION.CARDRIVER_ID", "$param.CarDriverId_param");
result.string(cond.toString());
```

This code means: If Parameter CarDriverId_param is "filled" (as it is in the Context CarDriver) then variable cond is, e.g., "CARRESERVATION.CARDRIVER_ID = '22fe825d-3899-4f1e-873f-f5d65b88e8b2'". (In the conditionProcess, the word "WHERE" is automatically added.) Then only those CARRESERVATION datasets are returned that show the CARDRIVER_ID handed over in the Parameter. If the Parameter is not filled (as it is in the Context CarReservation), then variable cond is empty, and no condition will be applied. In this case, *all* CARRESERVATION datasets are shown. (If you are interested to know in what SQL code this helper function results, simply log `cond.toString()`.)

You can log any value by using the methods of library `system.logging`:

Example code for using logging methods

```
var myVariable = "testValue";

// output in the Server log,
// see "Output" window of the Designer
logging.log(myVariable);

// output in the Client, via popup window
logging.show(myVariable);
```

Furthermore you can inspect variable values as well as the code processing via the Designer’s debugging functions. This is explained in the ADITO Designer Manual.

Now we can handle the Consumer side:

In the "Projects" windows, double-click on CarDriver_entity, so it will be opened in the Navigator window. Here, right-click on CarDriver_entity and choose "New Consumer". Name the Consumer "CarReservations". The Consumer will consequently appear under the node "Consumers". Set the connection to the Provider by editing the new Consumer’s properties:

- EntityName: Select CarReservation_entity.
- fieldName: Select the name of the Provider, i.e., "CarReservations".

Now, the Provider’s Parameter CarDriverId_param appears as sub-node of the Consumer. Click on this Parameter and edit its property "valueProcess" by inserting the following code:

CarDriver_entity.Consumers.CarReservations.CarDriverId_param.valueProcess

```
result.string(vars.get("$field.CARDRIVERID"));
```

The font color of the Parameter’s name will change from grey to white.

Now that we have established a dependency between CarDriver_entity and CarReservation_entity, we can select CarReservationFilter_view as a detail View of CarDriverMain_view: In the "Projects" window, double-click on CarDriverMain_view to open it in the Navigator window. Here, right-click on CarDriverMain_view and choose "Add reference to existing View..." in the context menu:

- EntityField: CarReservations (now available, due to the dependency!)
- View: CarReservationFilter_view

- Assign To: leave empty

Deploy, and open the CarDriver Context in the client. If you now select a driver and then press the "open" button, the CarDriverMain_view will open, with the CarDriverPreview_view to the left (a box, showing driver data) and the CarReservationFilter_view on the right, showing all reservations of this driver, in table form. (You will be able to test this once you have processed chapter "Example: Reserve this car", further below in this manual.)

10.4.1.2. Example: Showing all reservations of a car in the MainView

Now, with the knowledge obtained in the previous chapter, try it on your own: The task is to extend Context "Car" by a View showing all reservations of a specific car in the CarMain_view.

First, create a new Parameter for CarReservation_entity (Navigator window: right-click on CarReservation_entity > New Parameter), name it "CarId_param" (the Parameter name usually refers to the consuming EntityField) and set its property "expose" to true (this will make it accessible from outside CarReservation_entity). We need this Parameter in order to "hand over" the CARID of the selected car from the Context "Car" to the Context "CarReservation" (see later paragraph). Therefore, fill the valueProcess of this Parameter accordingly.

Establish a dependency via Provider/Consumer, as we had done in the previous example. Then, add CarReservationFilter_view as a reference to CarMain_view.

In the conditionProcess of the RecordContainer of CarReservation_entity, you can simply add a second condition for the Parameter referring to CARRESERVATION.CAR_ID:

CarReservation_entity.RecordContainers.db.conditionProcess

```
var cond = newWhereIfSet("CARRESERVATION.CARDRIVER_ID", "$param.CarDriverId_param")
.andIfSet("CARRESERVATION.CAR_ID", "$param.CarId_param");

result.string(cond);
```

If you are interested to know what SQL code this helper function produces, simply log `cond.toString()`.

10.5. Actions and ActionGroups

An Action in ADITO is an option to execute an arbitrary JDito code. Besides the predefined Actions (save, new, cancel, delete, etc.), you can specify further Actions according to your requirements.

An ActionGroup is an ADITO model to group Actions for being used in specific ViewTemplates, e.g., "Table". These ViewTemplates have special properties for referencing ActionGroups, usually named "favoriteActionGroup1", "favoriteActionGroup2", and "favoriteActionGroup3".

10.5.1. Configuration

Actions are assigned to an Entity, because they are always executed in an Entity’s Context.

To create and assign an Action, open an Entity in the Navigator window and choose option "New Action" in the Entity’s context menu. Then, enter a name in camelCase (see ADITO Information Document AID001, chapter "Spelling & Wording" > "ADITO models") and confirm with OK. The Action’s name will appear under a new node named "Actions". Click on the new Action and look at its properties:

- title: The text to be shown in the Actions menu (see below) or as Action button label.
- tooltip: The text to be shown as tooltip of the Action button.
- stateProcess: Code to specify cases when the Action should be in a specific state, e.g., disabled. By default, all Actions are available.
- onActionProcess: The actual code that will be executed when the user selects the Action.
- isMenuAction: see sub-chapter "Appearance"
- isObjectAction: see sub-chapter "Appearance"

10.5.2. Appearance

In ADITO xRM, Actions are usually shown in the following locations:

- In the PreviewView
  - Selectable via the three-dotted button in the PreviewView. This button is part of some ViewTemplates (in particular, the "Card"-type ViewTemplate) and usually includes the default Action "Delete".
  - As separate Action buttons (see properties "favoriteActionX" of some ViewTemplates, particularly "Card")
- In the FilterView, on the top of some ViewTemplates, e.g., "Table" (´see properties "favoriteActionGroupX"). Here, Actions can appear

    - as separate Action buttons;
    - clustered in ActionGroups;
    - included in the three-dotted button.

Where and when an Action appears, depends on the settings of the following properties:

- isMenuAction: defines if the Action is generally to be shown in the client. If this property is set to false, then the Action will appear *nowhere* in the client, whatever the other settings may be.
- isObjectAction: defines if the Action is to be displayed in relation to one specific dataset (selectable via the three-dotted button in the PreviewView)
- selectionType: see property description (bottom of property window)

In a FilterView, an Action is only shown, if it is part of an ActionGroup, and if this ActionGroup is specified as "favoriteActionGroup" in the property settings of the respective ViewTemplate (e.g., "Table"). This is independent from the state of property "selectionType".

Table 3. Visibility of Action buttons according to property settings

isMenuAction	**isObjectAction**	**selectionType**	**Visibility**
false	false	UNBOUND	nowhere
true	false	UNBOUND	as button via FilterView (only when specified via favoriteActionGroup)
false	true	UNBOUND	nowhere (isMenuAction = false)
false	false	MULTI	nowhere (isMenuAction = false)
true	true	UNBOUND	PreviewView (via three-dotted button), as button via FilterView (only when specified via favoriteActionGroup)
true	false	MULTI	as button via FilterView (only active, if at least 1 dataset is selected)
false	true	MULTI	nowhere (isMenuAction = false)

isMenuAction	isObjectAction	selectionType	Visibility
true	true	MULTI	PreviewView (via three-dotted button), as button via FilterView (only active, if at least 1 dataset is selected & if specified via favoriteActionGroup)

Here is an example of additional Actions in the "Card" ViewTemplate of the PreviewView, along with their settings:

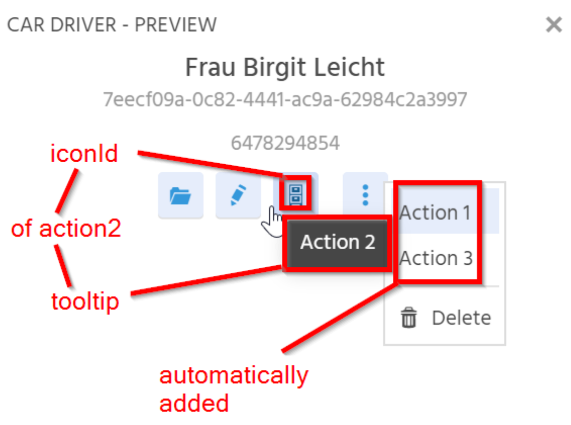


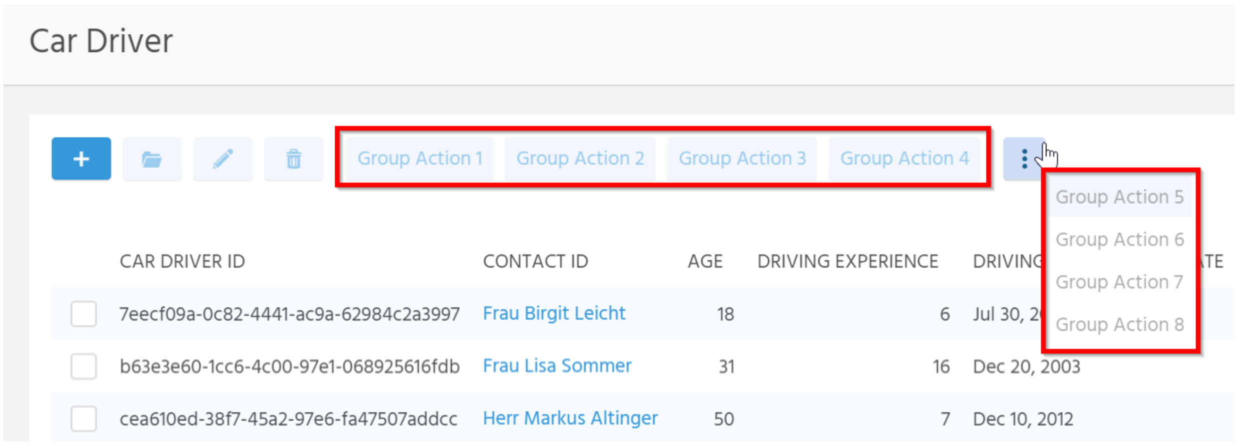
Figure 19. Example of additional Actions in the "Card" ViewTemplate

The property settings of all 3 Actions is, in this case:

- isMenuAction: true
- isObjectAction: true
- selectionType: MULTI

Furthermore, action2 is set in property "favoriteAction1" of the "Card" ViewTemplate of the PreviewView.

Here is another example, showing Actions that appear on the top of the "Table" ViewTemplate of the FilterView:


Car Driver

	CAR DRIVER ID	CONTACT ID	AGE	DRIVING EXPERIENCE	DRIVING
	7eecf09a-0c82-4441-ac9a-62984c2a3997	Frau Birgit Leicht	18	6	Jul 30, 2
	b63e3e60-1cc6-4c00-97e1-068925616fdb	Frau Lisa Sommer	31	16	Dec 20, 2003
	cea610ed-38f7-45a2-97e6-fa47507addcc	Herr Markus Altinger	50	7	Dec 10, 2012

Figure 20. Example of additional Actions in the "Table" ViewTemplate

All 8 Actions are included in an ActionGroup (Navigator > Entity > Context Menu > New Action Group):

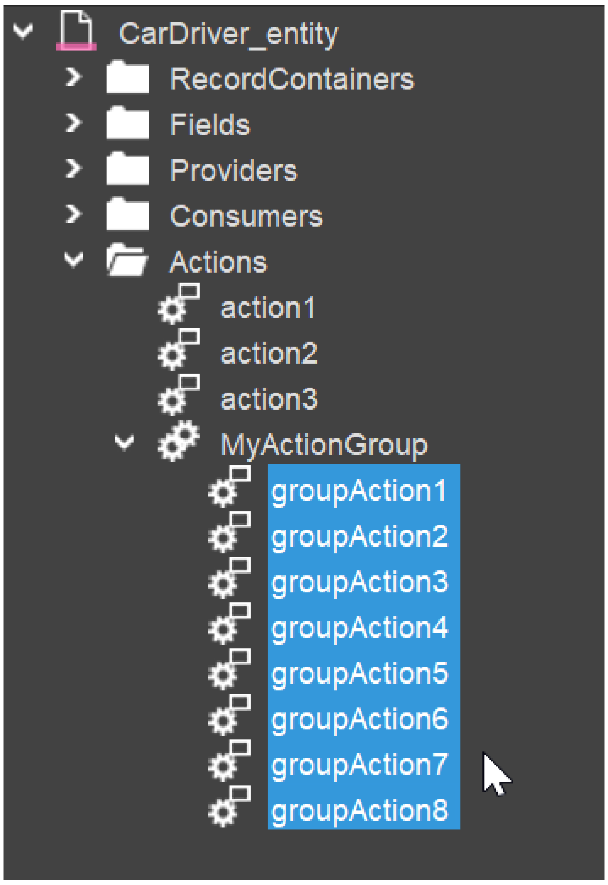


Figure 21. Example configuration of an ActionGroup

There are 2 ways to assign an Action to an ActionGroup:

1. Right-click on the ActionGroup and choose option "New Action" in the context menu.

   or

2. Create a new Action via the context menu of the Entity (Navigator > Entity > Context Menu > New Action) and then drag and drop it with the mouse pointer into the ActionGroup.

The property settings of all 8 Actions in the above example is as follows:

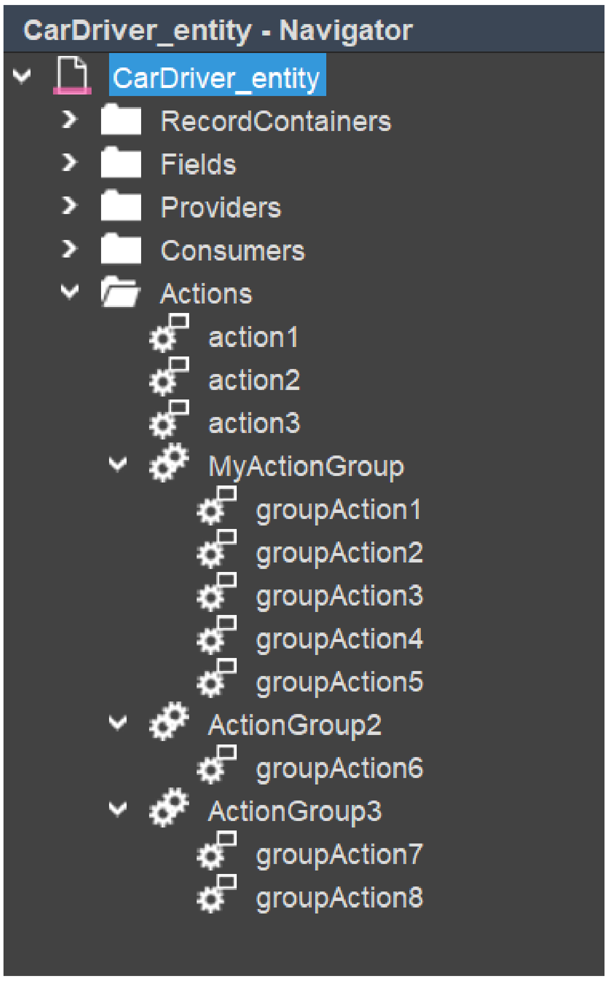
- isMenuAction: true
- isObjectAction: false
- selectionType: MULTI

Furthermore "MyActionGroup" is set in property "favoriteActionGroup1" of the "Table" ViewTemplate of the FilterView.

As you can see in the client screenshot, all 8 Actions are inactive. This is because property selectionType is set to MULTI, but none of the datasets is selected. As soon as one dataset is selected, all 8 Actions will become active.

You can also see that the first 4 Actions of the ActionGroup are displayed as separate buttons, while all further Actions are included in an Action list that appears if you click on the three-dotted button.

In the property sheet of some ViewTemplates, particularly those of type "Table", you can set up to 3 ActionGroups as "favoriteActionGroup". Please note: The display logic of property "favoriteActionGroup1" is different from the logic of properties "favoriteActionGroup2" and "favoriteActionGroup3". To watch the effect, please add 2 further ActionGroups and distribute the groupActions to these (using "Drag & Drop") as follows:


CarDriver_entity - Navigator

- CarDriver_entity
  - RecordContainers
  - Fields
  - Providers
  - Consumers
  - Actions
    - action1
    - action2
    - action3
    - MyActionGroup
      - groupAction1
      - groupAction2
      - groupAction3
      - groupAction4
      - groupAction5
    - ActionGroup2
      - groupAction6
    - ActionGroup3
      - groupAction7
      - groupAction8

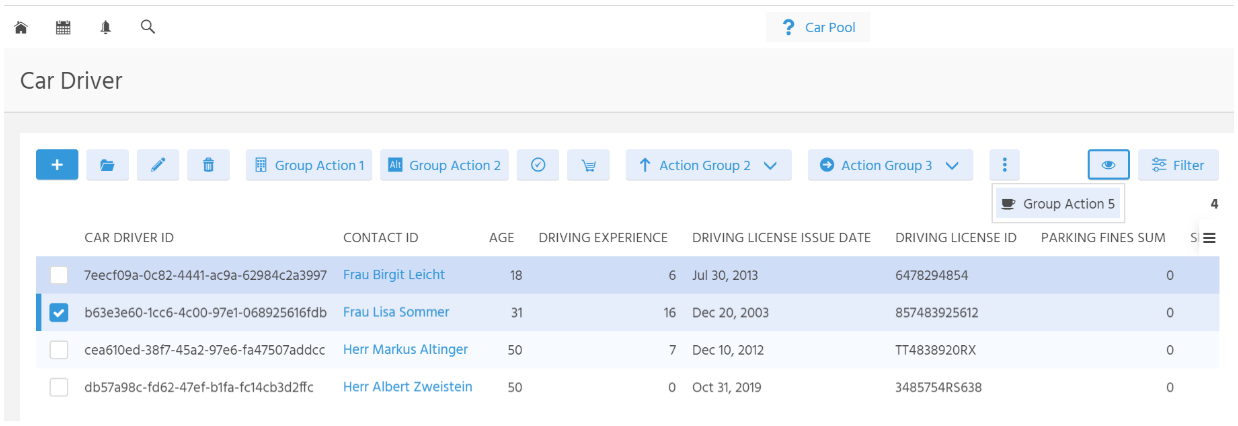
Figure 22. Example configuration of multiple ActionGroups

Furthermore, please set, separately for every Action and every ActionGroup, the following properties (enter any values you like):

- "title": In the client, this text will act as label (if the Action appears as separate button), or as Action list entry (if the Action appears in a combo box), respectively.
- "iconId" (choose any icon from the combo box): In the client, this icon will appear to the left of the title - or alone, if "title" is not set.

Setting an icon is especially important, if you view ADITO on a small screen, because in this case, the buttons are shown without "title" text (label). Thus, if you have not set an icon, you cannot identify the button, as it only appears as empty square.

After deploying, the result in the client should be something like this (select any dataset):



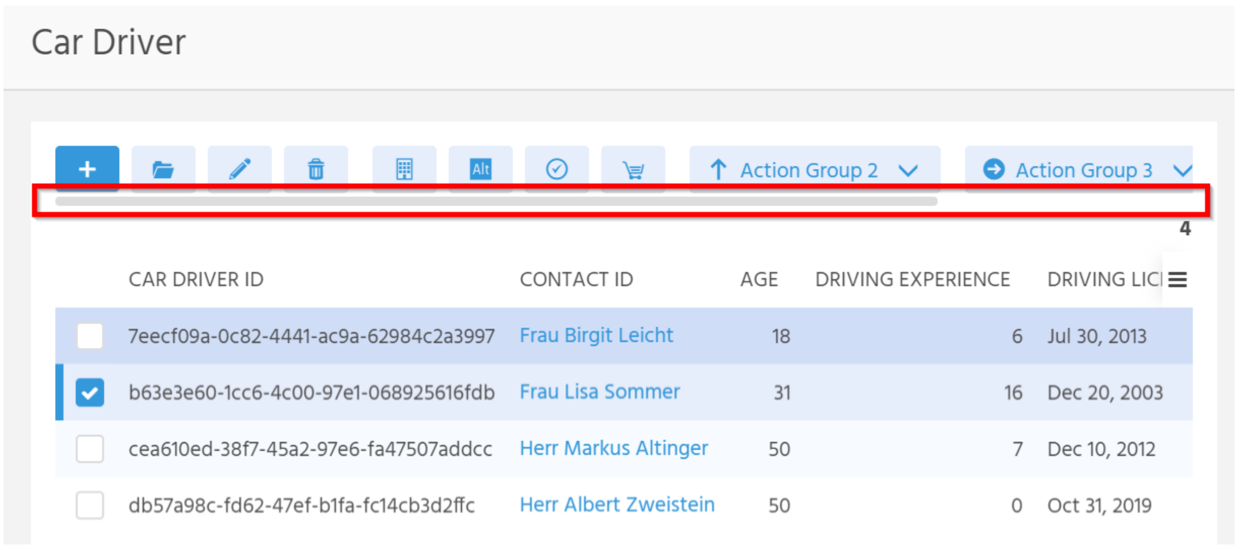
	CAR DRIVER ID	CONTACT ID	AGE	DRIVING EXPERIENCE	DRIVING LICENSE ISSUE DATE	DRIVING LICENSE ID	PARKING FINES SUM	S
☐	7eecf09a-0c82-4441-ac9a-62984c2a3997	Frau Birgit Leicht	18	6	Jul 30, 2013	6478294854	0	
✓	b63e3e60-1cc6-4c00-97e1-068925616fdb	Frau Lisa Sommer	31	16	Dec 20, 2003	857483925612	0	
☐	cea610ed-38f7-45a2-97e6-fa47507addcc	Herr Markus Altinger	50	7	Dec 10, 2012	TT4838920RX	0	
☐	db57a98c-fd62-47ef-b1fa-fc14cb3d2ffc	Herr Albert Zweistein	50	0	Oct 31, 2019	3485754RS638	0	

Figure 23. Example of how multiple ActionGroups appear in the client

As you can see, the display logic is as follows:

- "favoriteActionGroup1":
    - The first 4 Actions appear as separate buttons. If the screen is not wide enough, one up to all buttons are displayed without label, only with the Actions' icons (in this example, this is the case for groupAction3 and groupAction4).
    - All further Actions are included in an Action list that appears if you click on the three-dotted button.
- "favoriteActionGroup2" and "favoriteActionGroup3": All Actions of these ActionGroups are grouped in separate combo boxes, showing the respective ActionGroup’s title and icon.

If you watch this example on a very small screen (e.g., of a laptop), all buttons will be shown "abbreviated", i.e., without title, and a horizontal scroll bar is displayed in order to enable you to reach all Actions and further buttons (e.g., "Filter"):



	CAR DRIVER ID	CONTACT ID	AGE	DRIVING EXPERIENCE	DRIVING LICI
☐	7eecf09a-0c82-4441-ac9a-62984c2a3997	Frau Birgit Leicht	18	6	Jul 30, 2013
✓	b63e3e60-1cc6-4c00-97e1-068925616fdb	Frau Lisa Sommer	31	16	Dec 20, 2003
☐	cea610ed-38f7-45a2-97e6-fa47507addcc	Herr Markus Altinger	50	7	Dec 10, 2012
☐	db57a98c-fd62-47ef-b1fa-fc14cb3d2ffc	Herr Albert Zweistein	50	0	Oct 31, 2019

Figure 24. Example of ActionGroups and Actions to be displayed on a small screen (scroll bar appears)

Now, take some time and play around with these example Actions and their settings, in order to get familiar with the respective effects.

Then, let’s extend our car pool example project by some useful Actions:

10.5.3. Example: Reserve this car

When a car is selected, the user should have the option to reserve this car via an Action. The effect of the Action should be that the CarReservation Context is opened, with the respective car preselected.

First, we create a new Action for Car_entity, name it "reserveCar" (spelling convention: camelCase, starting with lowercase letter), and enter "Reserve this car" as property "title". After deploying, this Action will be immediately visible in the client, yet without effect.

Enter the following code in property "onActionProcess" of Action reserveCar:

Car_entity.reserveCar.onActionProcess

```
var params = {};

var carId = vars.get("$field.CARID");

if (carId) {
    params = {
       "CarId_param" : carId
    };
}

var recipe = neonFilter.createEntityRecordsRecipeBuilder().parameters(params).toString();

neon.openContextWithRecipe("CarReservation", null, recipe, neon.OPERATINGSTATE_NEW);
```

Explanations:

- The associative array `params` is used to carry information from the Context Car to the Context CarReservation, when the latter is opened with the dialog for creating new datasets. In this case, the only "workload" of `params` is CARID.
- If you want Context CarReservation to be opened in a new browser tab, you need to set method `openContextWithRecipe’s additional boolean parameter "pOpenInNewTab" to "true": `neon.openContextWithRecipe("CarReservation", null, recipe, neon.OPERATINGSTATE_NEW, null, true)`
  Just try it and watch the effect.

> Note that parameter "pOpenInNewTab" is only effective in desktop browsers.

> It will be ignored on tablets or mobile devices, and when targeting unsupported Views (e.g, PreviewViews or EditViews that have property "size" set to SMALL).

The above onActionProcess is the "outgoing" process, if you like.

Similar to the Provider/Consumer mechanism, we now need to "catch" this Parameter at the "ingoing" side, i.e., in the Context CarReservation. We need 2 separate processes: one for the CARID itself, and one for displaying specific features of the car identified by CARID.

> Using method `openContextWithRecipe` for "jumping" into another Context is not possible in the RecordContainer processes on(DB)`Insert`, on(DB)`Update`, and on(DB)`Delete`. This can lead to various errors.

CarReservation_entity.CAR_ID.valueProcess

```
if (vars.exists("$param.CarId_param"))
{
    result.string(vars.get("$param.CarId_param"));
}
```

Explanation:
Condition `if (vars.exists("$param.CarId_param"))` is only true, whenever Context Car is opened via the "reserveCar" Action. This if clause prevents overwriting the value of CAR_ID in readonly Views (where the Parameter does not exist).

Instead of CAR_ID, there should be displayed manufacturer, type, and color. We load this feature from the database with an SQL statement:

CarReservation_entity.CAR_ID.displayValueProcess

```
var carId = vars.get("$field.CAR_ID");

if (carId)
{
    var displayData = newSelect("MANUFACTURER, TYPE, COLOR")
    .from("CAR")
    .where("CAR.CARID", carId)
    .arrayRow();

    var carManufacturer = KeywordUtils.getViewValue($KeywordRegistryCarPool.carManufacturer(), displayData[0]);
    var type = displayData[1];
    var color = KeywordUtils.getViewValue($KeywordRegistryCarPool.carColor(), displayData[2]);

    result.string(carManufacturer + " " + type + " " + color);
}
```

Explanations:

- Using prepared statements via `SqlBuilder` is a safe way to execute an SQL statement. `arrayRow` specifies the return type of the SQL query: The contents of the first row is returned as a one-dimensional array (on the contrary, `arrayColumn` would return the contents of the first column as a one-dimensional array).
- The required values are retrieved by SQL. However, instead of writing something like
  `(…) select MANUFACTURER, TYPE, COLOR from CAR where CARID = carId (…)`
  we use prepared statements. The advantage of prepared statements is, amongst others, a higher security, because it prevents hacking by SQL injection.

10.5.4. Example: Reserve car for this driver

Now, use the previous example as a pattern for adding an Action that enables the reservation of a car for a given driver. I.e., we want to first select a driver in Context CarDriver and then "jump" to Context CarReservation, with the driver preselected. Try it on your own, before reading the solution.

Solution:

First, we create a new Action for CarDriver_entity, name it "reserveCarForDriver", and enter "Reserve car for this driver" as property "title". Make sure that the Parameter "CarDriverId_param" has been created, and its property "expose" has been set to "true" (see earlier chapter "Example: Showing all reservations of a driver in the MainView").
Now, we enter the following code in property "onActionProcess" of Action reserveCarForDriver:

CarDriver_entity.reserveCarForDriver.onActionProcess

```
import { neon, neonFilter, vars } from "@aditosoftware/jdito-types";

var params = {};
var carDriverId = vars.get("$field.CARDRIVERID");

if (carDriverId) {
    params = {
        "CarDriverId_param" : carDriverId
    };
}
var recipe = neonFilter.createEntityRecordsRecipeBuilder().parameters(params).toString();

neon.openContextWithRecipe("CarReservation", null, recipe, neon.OPERATINGSTATE_NEW,null,true);
```

We now need to "catch" this Parameter in the Context CarReservation:

CarReservation_entity.CARDRIVER_ID.valueProcess

```
if (vars.exists("$param.CarDriverId_param"))
{
     result.string(vars.get("$param.CarDriverId_param"));
}
```

Instead of CARDRIVER_ID, there should be displayed salutation, first name and last name of the driver . We load this feature from the database with an SQL statement:

CarReservation_entity.CARDRIVER_ID.displayValueProcess

```
var carDriverId = vars.get("$field.CARDRIVER_ID");

if (carDriverId)
{
    var displayData = newSelect("SALUTATION, FIRSTNAME, LASTNAME")
    .from("PERSON")
    .join("CONTACT", "CONTACT.PERSON_ID = PERSON.PERSONID")
    .join("CARDRIVER", "CARDRIVER.CONTACT_ID = CONTACT.CONTACTID")
    .where("CARDRIVER.CARDRIVERID", carDriverId)
    .arrayRow();

    var salutation = displayData[0];
    var firstname = displayData[1];
    var lastname = displayData[2];

    result.string(salutation + " " + firstname + " " + lastname);
}
```

Explanation:
As SALUTATION, FIRSTNAME, and LASTNAME are not fields of table CARDRIVER, but of table PERSON, we need a join in the SQL select statement.

Loading and writing datasets via `SqlBuilder` (or via the older methods `db.xxx`) ignores the permissions (access rights) configured by the client administrator! To load or write data respecting these permissions,

- set property "usePermissions" of the respective Entity/EntityFields to "true" (checkbox checked) **and**
- use the functionality of "LoadEntity" and "Write Entity" instead - see appendix LoadEntity and WriteEntity.

For further information on setting permissions please refer to the ADITO documentation for client administrators.

10.5.5. Multi Selection Action

To see an example of how a multi selection action can be implemented, take a look at the

`ChangeParticipantStatus_action` action of the `EventParticipant_entity` of the xRM Basic project.

In this example an additional context is used, `EventParticipantsChangeStatus`. It has it own view which serves as input. Its entity uses a dataless recordcontainer, contains one field which is linked via a consumer to the keyword entity to get the participant states. It also contains an action which will contain the change logic.

The action found in `EventParticipant` reads the current selection as recipe and puts in forward via a parameter into the additional context. The action of the `EventParticipantsChangeStatus` uses the recipe to get all ids of the selected records and writes the new status via the `SqlBuilder`. At the end of the action the current image (the view of `EventParticipantsChangeStatus`) is closed.

10.6. Calculated fields

The values of some EntityFields are the result of calculations, i.e., they are not only simply loaded from one single database column.

To assign a value to these kind of fields, there are 2 different ways:

- property "expression" of the EntityField’s RecordFieldMapping in the RecordContainer
- property "valueProcess" of the EntityField

These 2 variants will be explained below.

Calculated EntityFields are often readonly, i.e., they are not editable. In these cases, make sure that its property "state" is set to READONLY or DISABLED.

10.6.1. expression (RecordContainer)

The preferred way of calculating the value of an EntityField is property "expression" of the EntityField’s RecordFieldMapping. This property is defined with an SQL term e.g.

- a fix String or number
- the name of a database column or a combination of multiple database columns
- a sub-SELECT
- a complex calculation using specific SQL functions

Whenever an EntityField’s value has to be calculated, you should always first try to realize it this way, because the second way - using the valueProcess - can have serious performance-related impacts (see next chapter). Realizing a calculated EntityField via the "expression" property might include the challenge that you are not so familiar with programming complex algorithms via SQL, but still it will be worth to invest much effort in doing this, in order to avoid performance issues related to using the valueProcess (see next chapter).

The result of the code inserted in property "expression" will then be included in the "select" part of the SQL produced by the RecordContainer, e.g., when loading a table.

Example of code of property "expression" of an EntityField's RecordFieldMapping

```
var mySqlExpression = "'This is a test value.'";
result.string(mySqlExpression);
```

If you would log the SQL produced by the RecordContainer, you would see, for this example, something like this:

```
select MYCOLUMN1, MYCOLUM2, ('This is a test value.'), MYCOLUMN4, (...)
from MYTABLE
(...)
```

As you can see, ADITO automatically surrounds the result of the expression by round brackets, so you do not need to care for this when writing subselects, etc.

10.6.2. valueProcess (EntityField)

The second way of calculating the value of an EntityField is to use the EntityField's valueProcess. This property is defined with JDito code.

Example of code of property "valueProcess" of an EntityField

```
var myValue = "This is a test value.";
result.string(myValue);
```

10.6.2.1. Common use cases

The most common cases of using a valueProcess is

- to preset the value of an EntityField
  - when the EditView is opened in order to create new dataset;
  - dependent on the value of another EntityField.
- to calculate an EntityField's value, when a calculation is not possible via the expression of the RecordFieldMapping (see previous chapter). But, in this case, consider the following warning!

10.6.2.2. Warning

The same warning that was given for the displayValueProcess, also applies for the valueProcess of an EntityField. The following examples are only for demonstration purposes. Be aware that using the valueProcess of an EntityField can be a **performance killer**:
Whenever you want to calculate the value of an EntityField, using the value’s "expression" property in the RecordContainer usually enables a **higher performance** than other ways, in particular, than using property valueProcess of the EntityField (see below): The technical background is that the former simply extends the SQL statement of the loading process, while the latter initiates an additional loading process, which is executed separately **for every single dataset**.
However, if you want to create a new dataset that includes a predefined value for a specific field (or a combo box for selecting from a list of values), then setting "expression" is not enough, but you additionally need to set the valueProcess for this field.
To sum it up: Using the valueProcess of an EntityField can be a performance killer. Thus, as a rule of thumb, you should **always** try to calculate the value via SQL in the RecordContainer, using the "expression" property. If you need a value at creation time, you can additionally (not alternatively!) program a valueProcess. If both "expression" and valueProcess are filled, the ADITO system automatically ignores the valueProcess (but not the displayValueProcess!), whenever a value can be retrieved via "expression".
Find related information in appendix "Accessing the value of an EntityField".

10.6.2.3. Conditional execution

In most cases a valueProcess contains an "if" clause to define a condition for what code is to be executed, or if the valueProcess is to be executed at all. In this context, you need to consider that the valueProcess is triggered at various occasions (see following chapter). Thus, you should define the conditions for executing the respective code very precisely.

Example:
If you want to preset the value of an EntityField only in case it has not been set yet, the following code would be unsufficient, as an existing value would be overwritten:

```
result.string("preset subject text");
```

Instead, you need to included suitable conditions, like those in the following example:

```
if (vars.get("$sys.recordstate") == neon.OPERATINGSTATE_NEW &&
vars.get("$this.value") == null)
{
    result.string("preset subject text");
}
```

10.6.2.4. Trigger

The valueProcess is triggered at various occasions that cannot be listed completely, as there are many influencing factors like concurrency etc. Instead, these are the rules and conditions for the trigger.

When will the valueProcess of an EntityField principally be executed?

1. Depending on the operating state (`vars.get("$sys.operatingstate")`)
   a. VIEW
      In Operating state VIEW, ADITO tries to minimize the loading time by preferring to load the value via the RecordContainer. Therefore, the valueProcess is only executed, if
      i. the EntityField's value is available in the client, i.e., the EntityField is referenced in a ViewTemplate, **and**
      ii. the value of the EntityField has not already been loaded via the RecordContainer (e.g., via the RecordFieldMapping's property "expression").

         This principle is also true for the displayValueProcess.
   b. NEW/EDIT
      In the operating states NEW and EDIT, the valueProcess
      - will be executed for every relevant EntityField, i.e., in these operating states, more EntityFields are involved than in operating state VIEW: The valueProcess is executed for all EntityFields that are connected in the RecordContainer, i.e., that are being saved in the database - as well as all mandatory EntityFields.
      - is often used for presetting values or transforming values input by the user.
2. Dependencies between EntityFields:
   If the value of an EntityField is accessed in any process (e.g., but not only, in the valueProcess of another EntityField), the EntityField's value must be calculated under the actual conditions, which also can mean the execution of its valueProcess.

As performance problems are sometimes related to undiscovered executions of the valueProcess, it is generally helpful to activate JDitoLogging and track all executed processes. This will also reveal all occasions the valueProcess is triggered, which can

> be the basis for performance optimization. (Find more information in chapter Logging.)

10.6.2.5. Examples

In the xRM project, you can find a lot of examples of implementations of calculated EntityFields: Simply do a full-text search for the terms "expression" or "valueProcess" and scan the results.

Besides, we will now also extend our carpool example project by a few valueProcesses and expressions.

10.6.2.5.1. Example: Driving experience

In CarDriver_entity, the driving experience of a driver (field "drivingExperience") is calculated on the basis of the issue date of the driver’s driving license (field DRIVINGLICENSEISSUEDATE).

To display the driving experience in years, we enter the following code:

CarDriver_entity.drivingExperience.valueProcess

```
var drivingLicenseIssueDate = vars.get("$field.DRIVINGLICENSEISSUEDATE");

if (drivingLicenseIssueDate) {

    var drivingExperience = Math.floor(DateUtils.getDayDifference(drivingLicenseIssueDate)/365)  ;
    result.string(drivingExperience);
}
```

Explanation:
We calculate the driving experience (rounded to years), using the predefined function `DateUtils.getDayDifference` and dividing the result by 365 (which is simple but not 100% precise, of course - you might replace it by a better algorithm, if you like).

> The above example and the following examples are only for demonstration purposes. As explained before, you should always be aware that using the valueProcess of an EntityField can be a performance killer and that you should always prefer to realize calculation logic in the property "expression" of the corresponding record field, if this is possible. You may try this on your own, for the above example and for the below examples, if you like.

10.6.2.5.2. Example: Age

The age of a driver (CarDriver_entity’s field "age") is calculated on the basis of the value of Person_entity’s field DATEOFBIRTH, whose value is stored in the database field PERSON.DATEOFBIRTH. Now, as you have seen the preceding example and you know how to load fields from the database, try to write the code for the valueProcess on your own, before looking at the solution.

Solution:

To display the age, we enter the following code:

CarDriver_entity.age.valueProcess

```
var contactId = vars.get("$field.CONTACT_ID");

if (contactId) {

    var dateOfBirth = newSelect("DATEOFBIRTH")
    .from("PERSON")
    .join("CONTACT", "CONTACT.PERSON_ID = PERSON.PERSONID")
    .where("CONTACT.CONTACTID", contactId)
    .cell();

    if (dateOfBirth) {
       var age = Math.floor(DateUtils.getDayDifference(dateOfBirth)/365)  ;
       result.string(age);
    }
}
```

Explanations:
First, we retrieve the date of birth by a SQL select statement on table PERSON. Other than `arrayColumn()`, which we had used before, `cell()` returns one single database field. Then, we proceed the same as for field drivingExperience (see above).

As mentioned before, for performance reasons, you should avoid using a valueProcess or displayValue process without filling also the "value.expression" / "displayValue.expression" processes in the RecordContainer. Therefore, here is an example of how to calculate the age from a date of birth via SQL code.

The following example code refers to the xRM project’s standard Context "Person": Simply add an EntityField "age" to Person_entity and add this new field as new column of PersonFilter_view’s ViewTemplate "Persons". Then add the following code in the corresponding "expression" process:

Person_entity.db.age.value.expression

```
var sqlUtils = new SqlMaskingUtils();
var sql = sqlUtils.yearFromDate("CURRENT_DATE")
            + " - "
            + sqlUtils.yearFromDate("dateofbirth")
            + " - "
            + SqlBuilder.caseWhen(
                "("
                    + sqlUtils.monthFromDate("CURRENT_DATE")
                    + " < " + sqlUtils.monthFromDate("dateofbirth")
```

```
                      + "OR ("
                          + sqlUtils.monthFromDate("CURRENT_DATE")
                          + " = " + sqlUtils.monthFromDate("dateofbirth")
                          + " AND "
                          + sqlUtils.dayFromDate("CURRENT_DATE")
                          + " < " + sqlUtils.dayFromDate("dateofbirth")
                      + ")"
                  + ")"
              )
              .then("1")
              .elseValue("0")
              .toString();

result.string(sql);
```

After deploying, you will see the age displayed in the new column of PersonFilter_view (for each PERSON dataset that includes a DATEOFBIRTH value).

Now, try to use this code pattern for creating a similar suitable code for process CarDriver_entity.db.age.value.expression.

10.6.2.5.3. Example: Sum of fines

CarDriver_entity’s field "speedingFinesSum" is also a calculated field. It sums up all values of CarReservation_entity’s field "speedingFine", as far as referring to the given driver. Thus, we again need a database query for our valueProcess. Try it on your own, before looking at the solution.

Solution:

CarDriver_entity.speedingFinesSum.valueProcess

```
var carDriverId = vars.get("$field.CARDRIVERID");

if (carDriverId) {

    var speedingFinesArray = newSelect("SPEEDINGFINE")
    .from("CARRESERVATION")
    .where("CARRESERVATION.CARDRIVER_ID", carDriverId)
    .arrayColumn();

    var sum = 0.0;
    for (let i = 0; i < speedingFinesArray.length; i++) {
      if (speedingFinesArray[i]) {
        sum +=  parseFloat(speedingFinesArray[i]);
      }
    }
    result.string(sum);
}
```

Explanations:

- This time, we use function `arrayColumn()`, because we want to get all values of a column, SPEEDINGFINE. As `arrayColumn()` always returns an array of Strings (ignoring the data type of the database column), we must convert it into a decimal value (function `parseFloat`) before summing it up. For simplicity's sake, we ignore the currency here.
- The solution via a loop was only used to demonstrate function `arrayColumn()`. You will surely have noticed, that the better performance will be achieved by calculating the sum directly in the SQL statement. Try also this way now, by modifying the code. You may use function `cell()` in this case, as this returns only one single value, as string, not as array.
- `if (speedingFinesArray[i])` reflects the "implicit type conversion" functionality of JavaScript. Here, it is a simple way to avoid the sum becoming NaN, if a CARRESERVATION dataset shows no SPEEDINGFINE value - as, in this case, `speedingFinesArray[i]` is converted into the boolean value "false".

Of course, instead of calculating the sum via a "for" loop, you can also retrieve the sum via a modified SQL statement, in one step. Try it by yourself - but be aware that this might restrict your ADITO project to a specific database dialect (as SQL "sum" functionality differs between the dialects of the various database types).

Next, try to use your knowledge in order to write the code for the valueProcess of field "parkingTicketFinesSum" by yourself, before looking at the solution.

Solution:

CarDriver_entity.parkingTicketFinesSum.valueProcess

```
var carDriverId = vars.get("$field.CARDRIVERID");

if (carDriverId) {


    var parkingTicketFinesArray = newSelect("PARKINGTICKETFINE")
    .from("CARRESERVATION")
    .where("CARRESERVATION.CARDRIVER_ID", carDriverId)
    .arrayColumn();

    var sum = 0.0;
    for (let i = 0; i < parkingTicketFinesArray.length; i++) {
        if (parkingTicketFinesArray[i]) {
            sum +=  parseFloat(parkingTicketFinesArray[i]);
        }
    }
    result.string(sum);
}
```

10.6.2.5.4. Example: Sum of damages

Car_entity’s field "damages" is also a calculated field. It summarizes all texts of CarReservation_entity’s field "damage", as far as referring to the given car. Thus, we again need a database query for our valueProcess. Try it on your own, before looking at the solution.

Solution:

Car_entity.damages.valueProcess

```
var carId = vars.get("$field.CARID");

if (carId) {

    var damagesArray = newSelect("DAMAGE")
    .from("CARRESERVATION")
    .where("CARRESERVATION.CAR_ID", carId)
    .arrayColumn();

    var allDamages = damagesArray.join("; ");

    result.string(allDamages);
}
```

Furthermore, it is recommended to set property contentType of Car_entity’s field "damages" to LONG_TEXT, as this results in a scrollable text box shown in the client.

10.6.2.5.5. Example: Mileage

The value of Car_entity’s field mileage is the maximum value of CarReservation_entity’s field MILEAGERETURN, as far as the respective car is concerned. Try to write the required valueProcess on your own, before looking at the solution.

As always, there are multiple ways of solving the task - this is a suggestion:

Car_entity.mileage.valueProcess

```
var carId = vars.get("$field.CARID");

if (carId) {

    var maxMileage = newSelect("max(MILEAGERETURN)")
    .from("CARRESERVATION")
    .where("CARRESERVATION.CAR_ID", carId)
    .cell();

    result.string(maxMileage);
}
```

10.6.2.5.6. Example: carValue

The value of the car depends on its price and on its age. Per day, the value decreases by 0.03%. Furthermore, if the car has any damage, its value should be reduced by another 10%. Try to write the required valueProcess and displayValueProcess on your own, before looking at the solution.

Our suggested solution is:

Car_entity.carValue.valueProcess

```
var price = vars.get("$field.PRICE");
var dateOfManufacture = vars.get("$field.MANUFACTUREDATE");
var damages = vars.get("$field.damages");

if (price && dateOfManufacture) {
    var value = price;
    var ageInDays = Math.floor(DateUtils.getDayDifference(dateOfManufacture));
    // just a random percentage as example
    var valueLossPercent = -0.03;
    // random formula, similar to
    // calculation of "compound interest"
    value *= Math.pow((1+(valueLossPercent/100)), ageInDays);

    if (damages) {
        // just a random factor as example
        value *= 0.9;
    }
   result.string(value);
}
```

Car_entity.carValue.displayValueProcess

```
var carValue = vars.get("$this.value");
var currency = vars.get("$field.CURRENCY");

if (carValue && currency) {
 result.string(text.formatDouble(carValue, "#,##0.00") + " " + currency);
}
```

10.6.2.5.7. Example: Availability

Car_entity's field AVAILABILITY indicates whether or not a car is currently available, i.e., if it is currently lent (reserved) or not. Furthermore, a reservation should not be possible, if a car is not available in the requested timespan.

First try to write the required valueProcess on your own, before looking at the solution.

Our suggestion for the solution is:

Car_entity.availability.valueProcess

```
var carId = vars.get("$field.CARID");

if (carId) {

    var currentReservationId = newSelect("CARRESERVATIONID")
    .from("CARRESERVATION")
    .where("CARRESERVATION.CAR_ID", carId)
    .and("STARTDATE < CURRENT_TIMESTAMP")
    .and("ENDDATE > CURRENT_TIMESTAMP")
    .cell();

    var availability = "NO";
    if (currentReservationId == "") {
        availability = "YES";
    }
    result.string(availability);
}
```

If you are interested to know what SQL code this helper function returns, please refer to appendix "Database Access", chapter "SQL Helper Functions".

To decline a reservation of a car that is not available in the entered timespan, we use the property "onValidation" of CarReservation_entity. The code of this property will be executed, whenever the user enters a value of a field.

CarReservation_entity.onValidation

```
var carId = vars.get("$field.CAR_ID");
var startDate = vars.get("$field.STARTDATE");
var endDate = vars.get("$field.ENDDATE");
var reservationId = vars.get("$field.CARRESERVATIONID");

if (carId && startDate && endDate)
{
    var futureReservations = newSelect("CARRESERVATIONID, STARTDATE, ENDDATE")
    .from("CARRESERVATION")
    .where("CARRESERVATION.CAR_ID", carId)
    .and("ENDDATE > CURRENT_TIMESTAMP")
    .table();

    for (let i = 0; i < futureReservations.length; i++)
    {
        let aReservationId = futureReservations[i][0];
        let aStartDate = futureReservations[i][1];
        let aEndDate = futureReservations[i][2];
        if ((reservationId != aReservationId) && !(aStartDate > endDate || aEndDate < startDate))
        {
            result.string("Car is not available for the requested timespan.");
            break;
            }
        }
    }
```

Explanations:

- At first, we check if the user has selected a car as well as the start date and the end date of the reservation timespan.
- Then we load STARTDATE and ENDDATE of all reservations of the selected car, as far as they reach into the future (we do not need the past ones, as reservations will be done only for present or future)
- In a loop, we check if any of the existing reservations overlap with the timespan the user has selected (in fact, this "if" condition means "the opposite of no overlap"). If so, the result of the process is a text indicating that the car is not available for the requested timespan.
- If any text is given as result of the onValidation process, an automatism makes sure that
  - the validation is considered as "false";
  - the given text is displayed to the right of the "Save" button;
  - the "Save" button is disabled until the next call of the onValidation process.

> The onValidation process is one of several processes being executed when an Entity is processed. It is important to consider the order, in which these processes are internally handled. Find more information in appendix "Order of execution of Entity processes". **The onValidation process should only be used to validate data. Don’t react to changes here!**

> Loading and writing datasets via `SqlBuilder` (or via the older methods `db.xxx`) ignores the permissions (access rights) configured by the client administrator! To load or write data respecting these permissions,
>
> - set property "usePermissions" of the respective Entity/EntityFields to "true" (checkbox checked) **and**
> - use the functionality of "LoadEntity" and "Write Entity" instead - see appendix LoadEntity and WriteEntity.
>
> For further information on setting permissions please refer to the ADITO documentation for client administrators.

Now that we have configured the EntityField "availability", we can use it, e.g., in a ScoreCardViewTemplate that is placed as footer of the CarPreview_view (see chapter ScoreCard for further information):

- Open CarPreview_view in the Navigator window.
- Add a ViewTemplate of type "Score Card". In the "Add new ViewTemplate" dialog, choose "Assign to … footer".
- Configure the ViewTemplate’s properties as follows:
  - title: Can be left empty, as the title of the EntityField will be used automatically.
  - fields: Choose EntityField "availability"

After deploying and refreshing, the ScoreCardViewTemplate shows "YES" or "NO", depending on the actual availability. Now, we can optionally add a link to the ViewTemplate, in order to execute an Action:

- Create an Action named "availabilityAction"
- Un-check the checkbox of property "isObjectAction"
- Fill property onActionProcess with a code that shows a popup window including a sentence stating if the car is available or not:

```
var myMessage;

if (vars.get("$field.availability") == "YES") {
    myMessage = translate.text("This car is available.");
} else {
    myMessage = translate.text("This car is not available");
}
question.showMessage(myMessage);
```

- Now you can reference the new Action in property "fieldActions" of the ScoreCardViewTemplate.

After deploying and refreshing, hover over the ScoreCard with the mouse pointer: It has now got a link to the above Action, which opens a popup dialog showing a text.

10.7. AggregateFields

AggregateFields are ADITO models designed for aggregating EntityField values, e.g.

- counting datasets (e.g., via their ID column)
- summing up values
- showing minimum, maximum, or average values

In most cases, aggregations are related to a grouping.

A plain example of an implementation of AggregateFields is Context "Offer" in the ADITO xRM project. We will use this example to show the appearance of AggregateFields in the client and to explain how they can be configured in the ADITO Designer.

10.7.1. Appearance

In the client, navigate to Context "Offer" (menu group "Sales"). In the FilterView of this Context (OfferFilter_view), click on the view selection button (see upper right corner): Amongst other ViewTemplates, you can choose between 3 variants of ViewTemplates of type "DynamicMultiDataChart", which are all column charts, based on a preset grouping according to Offer_entity's EntityField STATUS (holding values like "Checked", "Open", "Sent", "Won"):

- **Count Chart** shows *how many* Offer datasets have which STATUS value.
- **Sum Chart** shows, separately for each STATUS value, the sum of all NET values (EntityField NET) of Offer datasets that have the respective STATUS value.
- **Probability Chart** shows, separately for each STATUS value, the average probabilty (EntityField PROBABILITY) of all Offer datasets that have a specific STATUS value.

Besides "DynamicMultiDataChart", there are further ViewTemplate types suitable for processing AggregateFields. Besides, AggregateFields can also be used in a Consumer. Find further information in chapter Properties allowing AggregateFields.

10.7.2. Configuration

10.7.2.1. when using DbRecordContainer

To configure an aggregate functionality, please proceed as follows:

1. Create an EntityField (and give it a title) as so-called "parent field" for the AggregateField (see below). As for the aggregate-related ViewTemplates of OfferFilter_view, these are Offer_entity's EntityFields COUNT (Count), NET (Total net), and PROBABILTY (Probability).

2. Add an AggregateField to the new EntityField (and give it a title): Right-click on the EntityField and choose option "New AggregateField" from the context menu. This will open a dialog requesting name and parent field.
    a. The name will be preset in the syntax "<EntityField name>_aggregate", which can be left unchanged ("COUNT_aggregate", "NET_aggregate", and "PROBABILITY_aggregate").
    b. The parent field is, by default, the EntityField to which the AggregateField has been added. In most cases, this can remain unchanged. (Exceptions are explained later.)
3. Open the RecordContainer node (e.g. "db") and double-click on the corresponding "xxx.value" RecordFieldMapping, e.g. COUNT_aggregate.value, in order to initialize it.
4. In the "Properties" window, you can now configure the properties
    a. recordField: Select the EntityField holding the values that are to be aggregated. Alternatively, you can specify an SQL code in property "expression". If both properties are set, the ADITO logic will exclusively use "expression" (same logic as for the RecordFieldMapping of an EntityField).
    b. aggregateType: Select the type of aggregation, e.g., COUNT, SUM, AVG (average), MIN(imum), or MAX(imum).
5. Create a new ViewTemplate that can process AggregateFields (see chapter Properties allowing AggregateFields). In our above example, these are ViewTemplates of type "DynamicMultiDataChart", which are assigned to OfferFilter_view.
6. Set the new ViewTemplate properties:
    a. title (e.g. "Sum chart")
    b. chartType (here: COLUMN)
    c. defaultGroupFields: As the scale of the x-axis is the result of a data grouping, enter the EntityField that determines the grouping in property "defaultGroupFields". In our example, it is the EntityField STATUS.
    d. columns: The values of the y-axis (= here: the height of the columns) are provided by an EntityField - in our 3 column chart examples, these are the EntityFields having AggregateFields assigned to: COUNT, NET, and PROBABILITY, respectively. Additionally, you need to set the corresponding AggregateField in columns aggregateEntityField, i.e., COUNT_aggregate, NET_aggregate, and PROBABILITY_aggregate, respectively.
    e. yAxisLabel: Optionally, you can set a label for the y-axis here.

10.7.2.2. when using JDitoRecordContainer

AggregateFields can also be applied when using a JDitoRecordContainer. The first steps are similar to when using a DbRecordContainer (see above).

A good pattern for learning the configuration of AggregateFields in a JDitoRecordContainer is the ADITO xRM project’s Context "Turnover". Proceed as follows:

- Login to an ADITO xRM project that includes demo data.
- In the global menu, click on "Sales forecast" (menu group "Sales"). This will open the Context that appears named "Turnover" in the ADITO Designer.
- Scan through the various ViewTemplates in the GroupLayout of this Context’s FilterView (internal name: "TurnoverDynamicMultiDataChart_view") and get familiar with the provided functionalities.
- In the Designer, navigate to Context "Turnover" and scan through its elements, e.g., inspect the configuration of the ViewTemplates assigned to TurnoverDynamicMultiDataChart_view.
- Read property "documentation" of Entity "Turnover_entity".
- Study this Entity’s configuration (EntityFields, RecordFieldMappings, AggregateFields etc.)
- Study the JDitoRecordContainers very carefully: Their "contentProcess" properties include a special grouping and result format.
    - The contentProcess of RecordContainer "jdito" has a comprehensive inline code documentation, which will help you to understand the technique.
    - Also study the configuration of JDitoRecordContainer "jditoDynamicMultiDataChart", as well as the code of its contentProcess.

10.7.3. displayValueProcess of an AggregateField

To format an AggregateField, you can use the displayValue or the displayValueProcess.

> AggregateFields currently can only be formatted by using the displayValueProcess. An expression exists, but as the system builds the aggregate from the return of the expression, it is not possible to use it for extra formatting or adding further elements to the result of the aggregate function.

Example of a displayValueProcess of an AggregateField

```
result.string("#: " + vars.get("$this.value"))
```

10.7.4. Usage in filter

If you want to use an AggregateField in a filter, set property "isFilterable" of the corresponding parentField’s RecordFieldMapping to true. It will then appear in the filter component (to the right of the FilterView) with the title of the AggregateField’s parent field (not of the AggregateField itself), so you can manually set a filter condition.

Furthermore, you can use AggregateFields also in the filterConditionProcess. Then, the following functions may be helpful:

- $local.isAggregateCondition: true, if the condition, for which the filterConditionProcess is executed, is based on an AggregateField; else false
- $local.conditionHaving: If a FilterExtension is executed due to an aggregation field, this variable returns the condition to be included in the subselect
- $local.columnPlaceholder: The value of the place holder, if you filter according to attributes und thus neet to retrieve the correct column name.

10.7.5. Usage in Consumer

AggregateFields can also be used in a Consumer, via its properties

- lookupIdField
- targetContextField
- targetIdField
- sortingField

If you want to show 2 aggregations in the same Entity, you need to use a second Consumer.

10.7.6. Properties allowing AggregateFields

Aggregate fields can be specified in various properties of the following ADITO models:

1. ViewTemplates
    a. properties "entityField" and "columns"/"fields":
        - Actions
        - CardTable
        - DynamicMultiDataChart
        - DynamicSingleDataChart
        - Gantt
        - Generic
        - GenericMultiple

        - ScoreCard
        - Table
        - TitledList
        - TreeTable

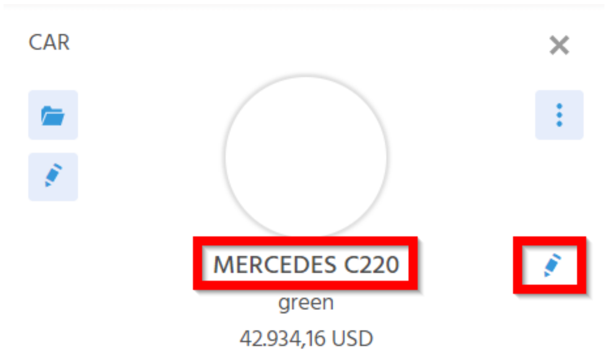
    b. ActionList: properties "titleField", "descriptionField", "iconField"

2. Consumer: properties "lookupIdField", "targetContextField", "targetIdField", "sortingField"

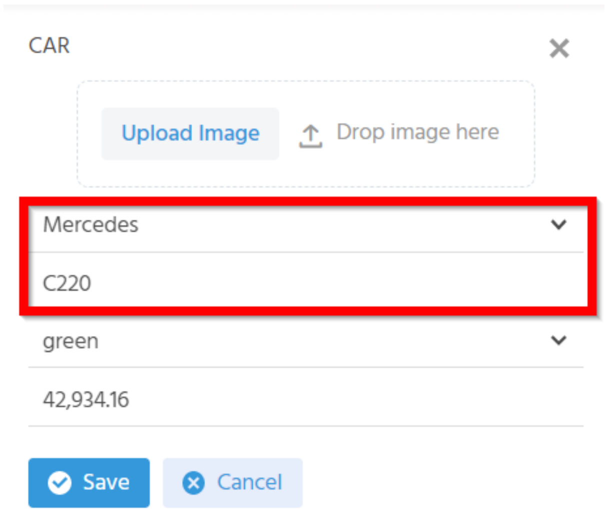
10.8. Field Groups

A Field Group is an ADITO model used for combining EntityFields, along with an additional configuration for editing.

Let’s use our car pool project for creating an example: Our task is to have both the EntityFields MANUFACTURER and TYPE set as property titleField of ViewTemplate CarPreviewCard (included in CarPreview_view):



When the edit button ("pencil") to the right is clicked, both EntityFields should be editable separately:



Configuration:

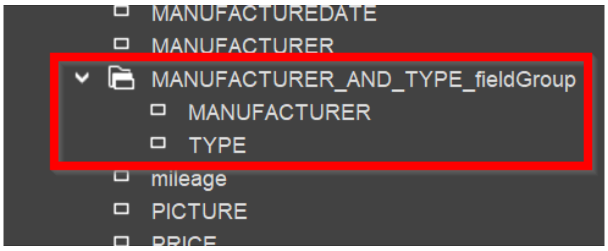
Open Car_entity in the Navigator window. Right-click on folder "Fields" and choose "New Field Group" from the context menu. In the following dialog, enter a suitable name, e.g., MANUFACTURER_AND_TYPE_fieldGroup.

In ViewTemplate CarPreviewCard, select the new Field Group in property titleField.

The central property of a FieldGroup is a valueProcess, in which you can combine any available EntityFields. The code for the above example is simple:

```
import { result, vars } from "@aditosoftware/jdito-types";

var manufacturer = vars.get("$field.MANUFACTURER");
var type = vars.get("$field.TYPE");

result.string(manufacturer + " " + type);
```

Making the EntityFields MANUFACTURER and TYPE both editable separately is even simpler: In the Navigator window, drag and drop these EntityFields on the FieldGroup, then they appear subordinated to the FieldGroup:



The 2 EntityFields will then be editable in this order, as soon as the client user clicks the edit button ("pencil") to the right.

Do not mistake the edit buttons: There is a further edit button to the *left* hand side of the CardViewTemplate. This button opens the EditView, while, in our case, we need the other edit button, which is to the *right* of the titleField. This edit button only refers to the EntityFields shown in the CardViewTemplate.

You may change the order of the editable EntityFields again by drag & drop in the Navigator window of the Designer.

If you click on the subordinated EntityFields in the Field Group, you see that no properties are shown. This is o.k., because the only purpose of these EntityField references is to determine, that, and in which order, the EntityFields can be edited in the client.

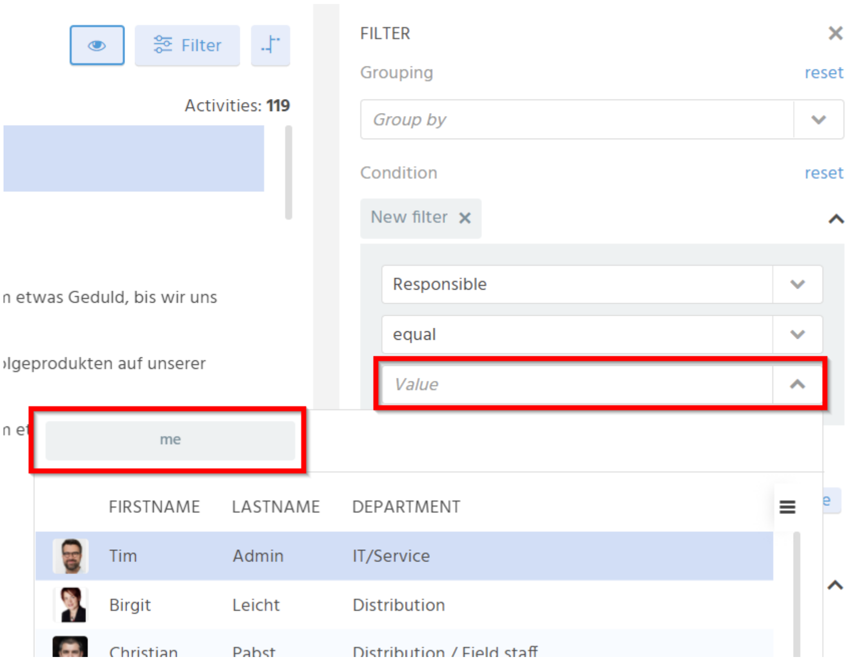
Advanced examples:

In the xRM project you can find various advanced examples of an implementation of a Field Group, e.g., FULL_NAME_fieldGroup, a Field Group of Person_entity. This Field Group combines the EntityFields SALUTATION, TITLE, FIRSTNAME, MIDDLENAME, and LASTNAME, as well as ORGANISATION_NAME, which is not editable. If you inspect the valueProcess of this FieldGroup you see that it references the Entity’s contentTitleProcess, which in turn is generated via the valueProcess of a separate EntityField named contenttitle.

10.9. Advanced filter options

10.9.1. Dynamic filter values

Besides fix filter values, ADITO enables the client user to specify also dynamic filter values. Look at an example included in the xRM project: In the filter component of ActivityFilter_view, you can filter according to property "Responsible". If you then open the "Value" combo box, you can not only select from the system's Employee records, but you can also choose value "me". This is a dynamic filter value, because it depends on the logged-in ADITO user (Employee).



	FIRSTNAME	LASTNAME	DEPARTMENT
	Tim	Admin	IT/Service
	Birgit	Leicht	Distribution
	Christian	Pabst	Distribution / Field staff

Figure 25. Example of dynamic filter value "me" (Context "Activity")

In the Designer, this feature is configured as follows:

Activity_entity's EntityField RESPONSIBLE has a Consumer "Employees", which depends on Employee_entity's Provider "Employees". This Provider's property "filterVariablesProcess" includes the core code of the feature:

Employee_entity.Employees.filterVariablesProcess

```
var res = {
    "global.user.contactId": translate.text(
"${FILTER_DYNAMIC_VALUE_ME}")
};

result.object(res);
```

The result of the filterVariablesProcess can be an arbitrary number of key-value pairs, with

- key being the variable whose value is to be used
- value being the display value to be used in the client

In the above example, the result means "Show the text 'me' as additional filter value. If the client user selects it, use the value of variable `$global.user.contactId` as filter value."

This principle can be used with any $global und $sys variable. If required for this purpose, further $sys variables can be ordered from ADITO’s development department. But you can also define your own $global variables via `vars.set("$<global.xxx.yyy>", variableValue)` in process "autostartNeon".

The technical background, why this feature is available in the Provider model is:
As you can (optionally) specify a lookupIdfield in the Provider in order to use a different UID, it must be possible to filter also according to different values.

Hypothetical example:

- UID: CONTACTID
- lookupIdfield of Provider "#PROVIDER" (the UID will be used)
  → In this case, a dynamic filter "my company" requires the **CONTACTID** of the user’s company.
- lookupIdfield of Provider "XYZ": ORGANISATIONID
  → In this case, a dynamic filter "my company" requires the **ORGANISATIONID** of the user’s company.

10.9.2. Filter presets

Besides filters set by the client user, it is also possible to include customized filter presets, e.g., to be automatically executed when a Context is opened.

10.9.2.1. FilterBuilder

JDito includes the FilterBuilder, which is a builder pattern that allows you to define/configure even complex filters quite comfortably and intuitively. (In earlier versions you had to realize this via a big JSON string, created manually.)

To use this functionality, you first need to import the system module `neonFilter`. It includes the methods

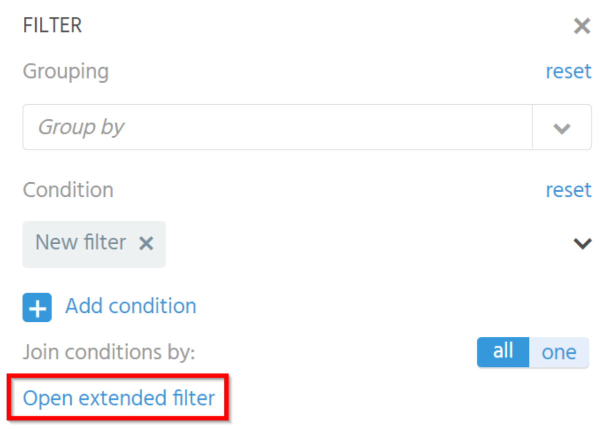
- `createFilterGroup`, which creates an object of class `FilterGroup`
- `createFilterCondition`, which creates an object of class `FilterCondition`

You can create multiple instances of these objects. Each can be configured in detail and then nested -

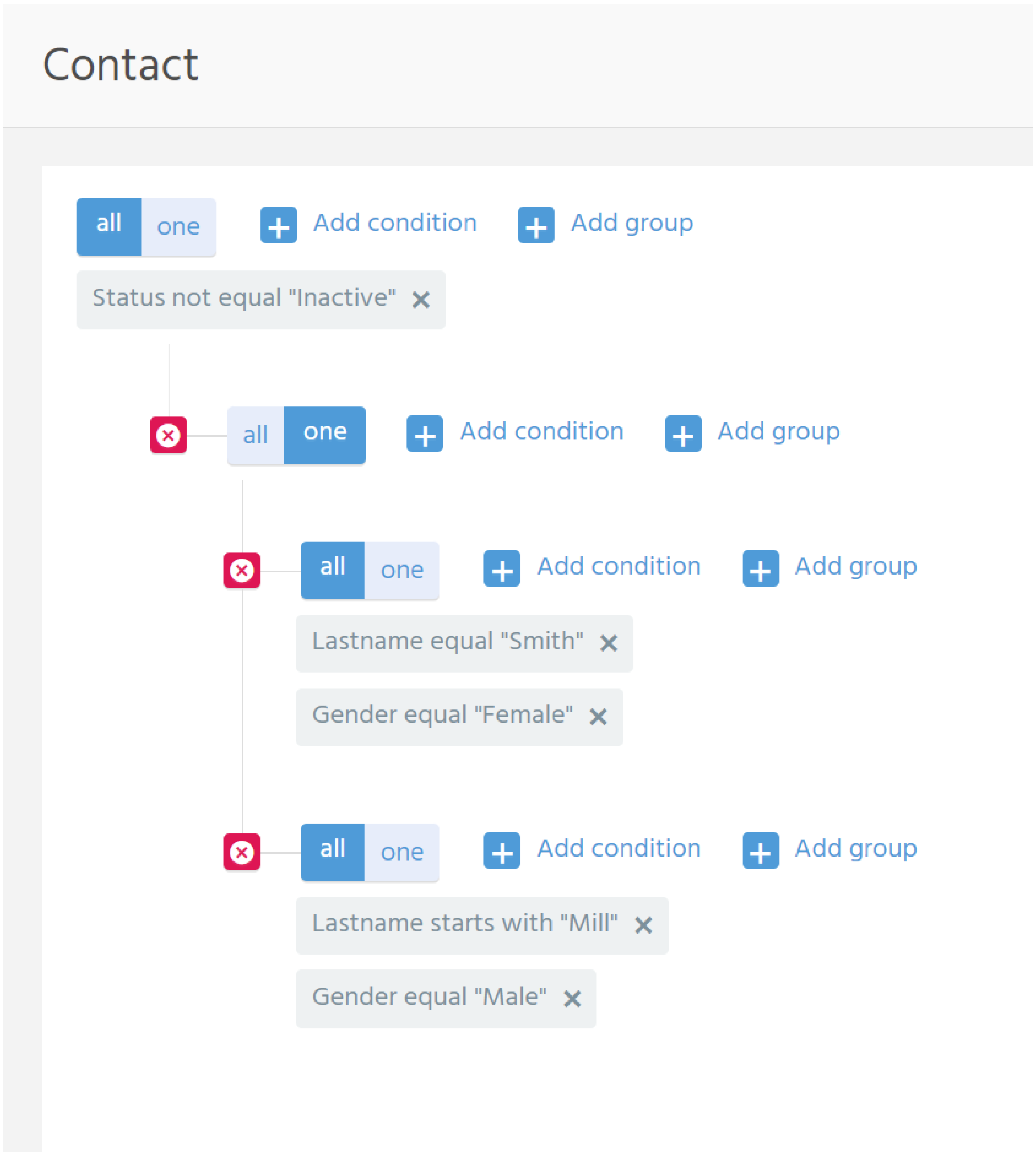
which finally results in an (extended) filter.

The handling of these methods and objects is similar to the usage of the "extended Filter" in the client. Therefore, we will create an arbitrary example of an extended filter in the client and then learn how to configure the same filter in JDito.

First, click on "Open extended filter" in the FilterView of Context "Contact".



Here is an example of an extended filter, containing 4 filter groups (represented by the lines starting with the all/one switch), each with 0 to 2 filter conditions (represented by the lines with grey background).


Contact

all one Add condition Add group

Status not equal "Inactive"

all one Add condition Add group

all one Add condition Add group

Lastname equal "Smith"

Gender equal "Female"

all one Add condition Add group

Lastname starts with "Mill"

Gender equal "Male"

The effect of this filter is to show all "Contacts" (= datasets of Person_entity) that are

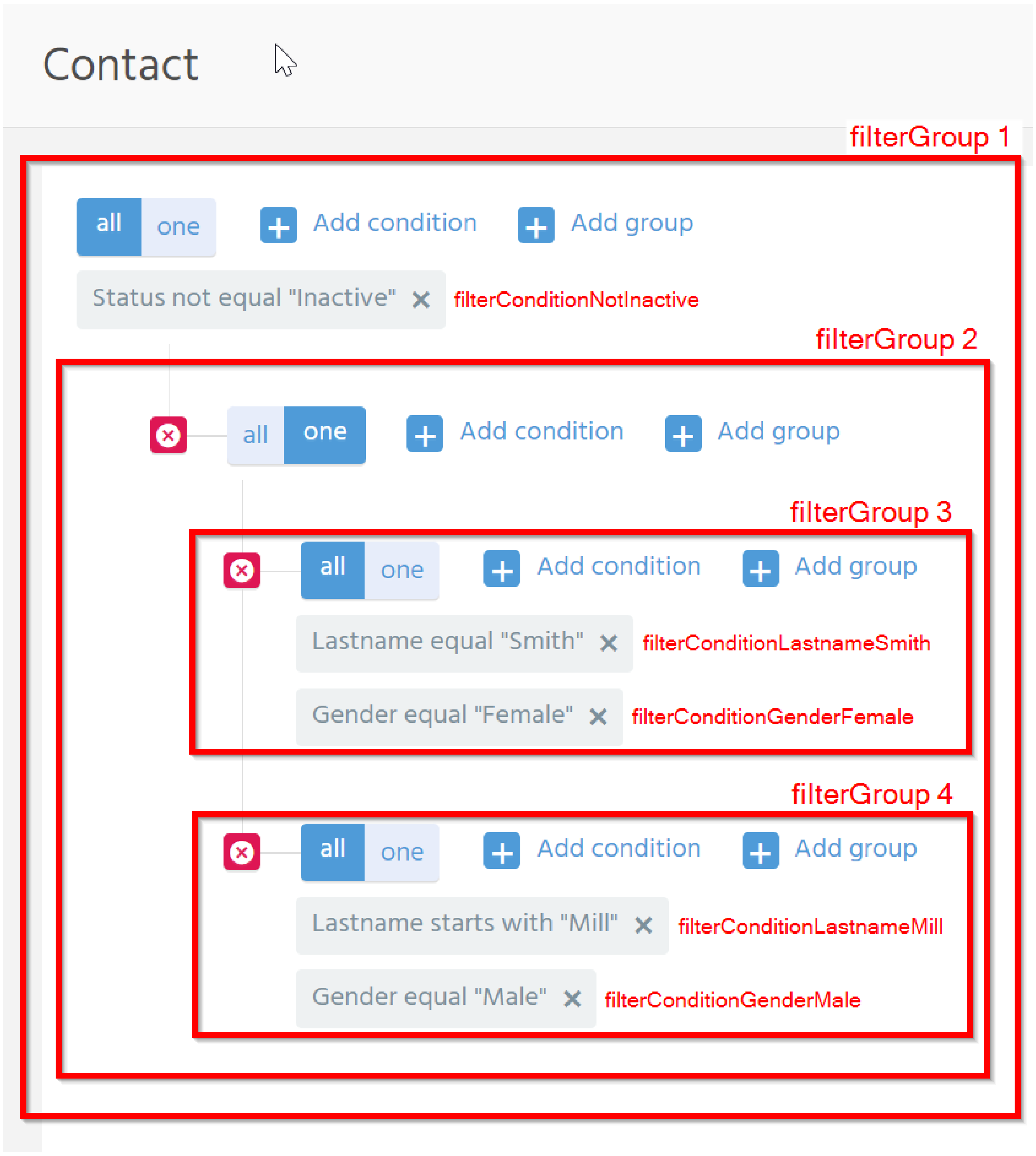
- not inactive AND
    - EITHER female and having the last name "Smith"
    - OR male and having a last name starting with "Mill"

Now, this is how the same filter is configured in JDito (cf. labeled screenshot below):

Examples of how to use the filter builder pattern

```
var filterGroup1 = neonFilter.createFilterGroup();
filterGroup1.mergeOperator(neonFilter.MERGE_OPERATOR_AND);

var filterConditionNotInactive = neonFilter.createFilterCondition()
.field("STATUS")
.searchOperator(neonFilter.SEARCH_OPERATOR_NOT_EQUAL)
.contentType(neonFilter.CONTENT_TYPE_TEXT)
.value("Inactive")
.key("CONTACTSTATINACTIVE");

filterGroup1.addFilterCondition(filterConditionNotInactive);

var filterGroup2 = neonFilter.createFilterGroup();
filterGroup2.mergeOperator(neonFilter.MERGE_OPERATOR_OR);
filterGroup1.addFilterGroup(filterGroup2);

var filterGroup3 = neonFilter.createFilterGroup();
filterGroup3.mergeOperator(neonFilter.MERGE_OPERATOR_AND);
filterGroup2.addFilterGroup(filterGroup3);

var filterConditionLastnameSmith = neonFilter.
createFilterCondition()
.field("LASTNAME")
.searchOperator(neonFilter.SEARCH_OPERATOR_EQUAL)
.contentType(neonFilter.CONTENT_TYPE_TEXT)
.value("Smith")
.key("Smith");

filterGroup3.addFilterCondition(filterConditionLastnameSmith);

var filterConditionGenderFemale = neonFilter.createFilterCondition()
.field("GENDER")
.searchOperator(neonFilter.SEARCH_OPERATOR_EQUAL)
.contentType(neonFilter.CONTENT_TYPE_TEXT)
.value("Female")
.key("f");

filterGroup3.addFilterCondition(filterConditionGenderFemale);

var filterGroup4 = neonFilter.createFilterGroup();
filterGroup4.mergeOperator(neonFilter.MERGE_OPERATOR_AND);
filterGroup2.addFilterGroup(filterGroup4);

var filterConditionLastnameMill = neonFilter.createFilterCondition()
.field("LASTNAME")
.searchOperator(neonFilter.SEARCH_OPERATOR_STARTSWITH)
.contentType(neonFilter.CONTENT_TYPE_TEXT)
```

```
.value("Mill")
.key("Mill");

filterGroup4.addFilterCondition(filterConditionLastnameMill);

var filterConditionGenderMale = neonFilter.createFilterCondition()
.field("GENDER")
.searchOperator(neonFilter.SEARCH_OPERATOR_EQUAL)
.contentType(neonFilter.CONTENT_TYPE_TEXT)
.value("Male")
.key("m");

filterGroup4.addFilterCondition(filterConditionGenderMale);
```



Contact

filterGroup 1

all one Add condition Add group

Status not equal "Inactive" filterConditionNotInactive

filterGroup 2

all one Add condition Add group

filterGroup 3

all one Add condition Add group

Lastname equal "Smith" filterConditionLastnameSmith

Gender equal "Female" filterConditionGenderFemale

filterGroup 4

all one Add condition Add group

Lastname starts with "Mill" filterConditionLastnameMill

Gender equal "Male" filterConditionGenderMale

The above code is simplified. In practice, of course, you would reference KeywordEntries not directly, but via the usual JDito methods, e.g.,

- `.key($KeywordRegistry.contactStatus$inactive())` instead of `.key("CONTACTSTATINACTIVE")`
- `.value(KeywordUtils.getViewValue($KeywordRegistry.contactStatus(), $KeywordRegistry.contactStatus$inactive()))`

> instead of `.value("Inactive")`

As you can see, creating a FilterGroup is always the starting point. "Into" this FilterGroup, you can add ("nest")

- either a FilterCondition
- or another FilterGroup, which in turn can have FilterGroups or FilterConditions added ("nested")

This "nesting" of FilterGroups and other FilterGroups or FilterConditions can, in theory, be continued up to an unlimited depth (with limits as for performance, of course).

The methods mean:

- FilterGroup:
  - `.mergeOperator`: The operator (mathematical term: "logical connective") to be applied for connecting the filter group’s conditions and subordinated filter groups. Possible values: "and", "or". The parameter should be specified by using the respective constants given in `neonFilter`, e.g., `neonFilter.MERGE_OPERATOR_AND`.
  - `.addFilterGroup`: "Nests" one filter group "into" another. The subordinated filter group’s result will be connected with the filter conditions using the specified `mergeOperator` (see above).
  - `.addFilterCondition`: Specifies a filter condition to be applied in this filter group. This filter condition will be connected with further filter conditions as well as with subordinated filter groups' results using the specified `mergeOperator` (see above).
  - `.toJson()`: Converts the filter group and all its nested content into a big JSON string. In earlier ADITO versions, this was required for passing the filter to specific JDito methods that did not yet accept the filter object as argument, but only a JSON string - e.g., `neon.setFilter`. But now, in most cases, you do not need to convert the filter into a JSON string, because methods like `neon.setFilter` accept the FilterGroup itself as argument. Also process properties like initFilterProcess (see below), accept the FilterGroup to be passed as result, without the need to convert it into a JSON string first: `result.string(<myFilterGroup>)` (but do NOT use `result.object(<myFilterGroup>)` in these cases).
- FilterCondition:
  - `.field`: The EntityField the filter condition refers to
  - `.contentType`: The contentType of the EntityField. The parameter should be specified by using the respective constants given in `neonFilter`, e.g., `neonFilter.CONTENT_TYPE_TEXT`.

◦ `.searchOperator`: The search operator (relational operator) to be applied for the comparison between the values of the EntityField and the value given in method `.key`. Examples: "greater than", "equals", "starts with". The parameter should be specified by using the respective constants given in `neonFilter`, e.g., `neonFilter.SEARCH_OPERATOR_EQUAL`.

◦ `.key`: The value to compare the EntityField’s values with when the filter is executed. This value is NOT displayed in the client.

◦ `.value`: The value to be **displayed** in the client when the filter has been set (NOT used for filtering, only for displaying purposes!) If there is no difference between the values of `.key` and `.value`, then `.value` must be specified anyway - simply repeating the value of `.key`(see, e.g., the above example "filterConditionLastnameSmith") .

10.9.2.2. initFilterProcess

If you want a specific filter to be preset when a Context is opened (i.e., when calling the FilterView), the most common way is to use the Entity’s initFilterProcess. The required result can easily be configured using the FilterBuilder (see previous chapter).

Here is an example task, which can be used as pattern: When opening the FilterView of Context Car, we want to see only cars, whose license plate number includes the letter "M". This requires the following code:

Car_entity.initFilterProcess

```
import { neon, neonFilter, result, vars } from "@aditosoftware/jdito-types";

if (vars.get("$sys.presentationmode") === neon.CONTEXT_PRESENTATIONMODE_FILTER)
{
    var recordState = vars.get("$sys.recordstate");
    if (recordState != neon.OPERATINGSTATE_SEARCH)
    {
        var filter = neonFilter.createFilterGroup()
        .mergeOperator(neonFilter.MERGE_OPERATOR_AND)
        .addFilterCondition(neonFilter.createFilterCondition()
            .field("LICENSEPLATENUMBER")
            .key("M")
            .value("M")
            .searchOperator(neonFilter.SEARCH_OPERATOR_CONTAINS)
            .contentType(vars.get("$property.LICENSEPLATENUMBER.contentType"))
        );
    result.string(filter.toString());
    }
}
```

Find more information about the difference between "operating state" and "record

> state" in the appendix "Operating state vs. record state".

Here is another example, from the xRM project. This code makes sure that only "active" company datasets are shown:

Organisation_entity.initFilterProcess (fragment)

```
var filter;

(...)

if (vars.get("$sys.presentationmode") === neon.CONTEXT_PRESENTATIONMODE_FILTER)
{
    var statusInactive = $KeywordRegistry.contactStatus$inactive();

    filter = neonFilter.createFilterGroup()
        .mergeOperator(neonFilter.MERGE_OPERATOR_AND)
        .addFilterCondition(neonFilter.createFilterCondition()
            .field("STATUS")
            .key(statusInactive)
            .value(KeywordUtils.getViewValue($KeywordRegistry.contactStatus(), statusInactive))
            .searchOperator(neonFilter.SEARCH_OPERATOR_NOT_EQUAL)
            .contentType(neonFilter.CONTENT_TYPE_TEXT)
            );
}

result.string(filter);
```

Filter referencing a FilterExtension:

If your filter condition is related to a FilterExtension (see chapter "FilterExtension"), the syntax of the argument of method `field` is like this:

```
"#EXTENSION.TestfilterExtension.TestfilterExtension#TEXT"
```

Here is a code fragment as an example:

Example of referencing a FilterExtension in a filter condition

```
(...)
var filterCondition =  neonFilter.createFilterCondition()
   .field("#EXTENSION.TestfilterExtension.TestfilterExtension#TEXT")
   .key(companyId)
   .value("13")
   .searchOperator(neonFilter.SEARCH_OPERATOR_EQUAL)
   .contentType(neonFilter.CONTENT_TYPE_TEXT);
(...)
```

10.9.2.3. neon.setFilter

If you want a specific filter to be applied in another part of your project - e.g., in an Action - you need to use method `neon.setFilter`:

Pattern for setting a filter in JDito

```
var filter = neonFilter.createFilterGroup();

(...) // configuration of filter - cf. above examples

neon.setFilter("#ENTITY", filter);
```

(In earlier ADITO versions, `neon.setFilter`required a JSON string as argument, but now you can simply pass the filter builderpattern (FilterGroup) as shown in the example.)

10.9.3. FilterExtension

FilterExtension is an ADITO model used for extending the standard filter (which are controlled via the "isFilterable" properties of the RecordFieldMappings in the RecordContainer) by an additional filter option - e.g., a filter criteria that is related to a field of another Entity.

Examples in ADITO xRM:

- "Favorites_filter", a FilterExtension included in the RecordContainers of several Entities, e.g., of Organisation_entity.
- "Phase_filter" ("Phase_filterExtention"), a FilterExtension included in the RecordContainer of Salesproject_entity.

To understand these examples, you might first study the following paragraphs.

10.9.3.1. General example

Generally, a FilterExtension can be added as follows:

10.9.3.1.1. Creating a new FilterExtension

Open your Entity in the Navigator, right-click on its RecordContainer, and choose "Add Filter Extension" from the context menu. Enter a name of your choice, e.g., "hasMyEntityFieldSet".

Now, fill the new FilterExtension's properties as follows:

10.9.3.1.2. General properties

- "title": Enter the text of the respective list item to appear in the filter's combo box "Property", e.g. "Has my EntityField set".

- "contentType": Enter the data type of the values to be entered or selected in the filter's field "Value". This data type will then, amongst others, determine the list of relational operators from which the user can select in the filter component's field "Operator". For our example, the contentType should be set to "TEXT".

10.9.3.1.3. filterValuesProcess

Fill property filterValuesProcess only if you want to select the values (in the filter's field "Value") via a combo box. The result of this property's process must be an array of value pairs:

Example of data structure of filterValuesProcess result

```
(...)
var myFilterValues = [];
myFilterValues.push(["myID1", "mycombo boxListItem1"]);
myFilterValues.push(["myID2", "mycombo boxListItem2"]);
myFilterValues.push(["myID3", "mycombo boxListItem3"]);
(...)
result.object(myFilterValues);
```

The second values of the value pairs are displayed in the combo box, while the first values are to evaluate the selection in the filterConditionProcess (see below); often this first value is a key (e.g., the UID of a dataset).
In many cases, the above result array is generated via an SQL selection:

Example of filterValuesProcess using an SQL selection

```
var myFilterValues = newSelect("MYIDCOLUMN, MYCOLUMNFORCOMBOBOXLISTITEMS")
                        .from("MYTABLE")
                        .table();

result.object(myFilterValues);
```

If, however, you want to enter the value as free text (no selection via combo box), then property filterValuesProcess must remain in default status (empty).

The specific code of the filterValuesProcess suitable for the carpool example will be added in a future version of this manual. You may try it by yourself meanwhile.

10.9.3.1.4. useConsumer/consumer

Checking property useConsumer enables you to use lookup functionality via a Consumer. It is an alternative to specifying a filterValuesProcess: As soon as you have checked this checkbox, property filterValuesProcess disappears and property "consumer" appears instead. Here, you can select a Consumer that delivers the selectable filter values, with the corresponding Provider Entity's

- contentTitleProcess or (if set) LookupView to be used for displaying the selectable values in the combo box;
- UID column (= its primary key) acting as value to be evaluated in the filterConditionProcess (see chapter below)

If you uncheck property useConsumer again, property filterValuesProcess (and its value) will reappear instead of property "consumer". The values of these alternative properties will always remain stored, independently from the value of property useConsumer.

10.9.3.1.5. filterConditionProcess

The property filterConditionProcess is used for reacting to the value the user has input/selected in the "Value" field of the filter component. The result of this process is an SQL condition that will be added to the conditionProcess of the RecordContainer (i.e., to the "WHERE" part of the SQL code used for filling the FilterView). Like in the conditionProcess, the SQL code words "WHERE" and "AND" must not be included.

The following variables are available in the filterConditionProcess:

- The user's input/selection can be retrieved via the local variable "rawvalue": `vars.get("$local.rawvalue")` This variable holds
  - the text typed in by the user, if the input was done via a free text field (see above, filterValuesProcess)
  - the first value of the value pair corresponding to the user's selection, if the input was done via a combo box (see above, filterValuesProcess)

    Furthermore, there are 2 $local variables to retrieve the relational operator that the user has selected in the "Operator" field of the filter component:
- "comparison": `vars.get("$local.comparison")` This variable holds a String value representation of the selected operator, e.g. "EQUAL", "CONTAINS", or "STARTSWITH". This operator selection can then be evaluated via "if" clauses or via "switch/case" (here, you can, if required, use the `SqlBuilder.XXX` functions, e.g., `SqlBuilder.EQUAL()` - see example code below).
  Possible values of `$local.comparison` are EQUAL, GREATER, LESS, GREATER_OR_EQUAL, LESS_OR_EQUAL, NOT_EQUAL, CONTAINS, CONTAINSNOT, STARTSWITH, ENDSWITH, ISNULL, and ISNOTNULL. Furthermore, there is a value named EQUAL_ANY, if there is a comparison with multiple values - meaning that *at least one* of the values matches (in contrast to EQUAL,

meaning that *all* elements match). Value NONE means that there is no operator.

- "operator": `vars.get("$local.operator")` This variable holds an Integer value representation, deduced from the selected operator. However, it is not unique for every operator (some operator selections result in equal Integer values). Experienced users can use this value to simplify the evaluation of the selected operator, especially in SQL statements. But do not be confused by this variable: You can make use of the full FilterExtension's functionality by ignoring variable "operator" and evaluate only variable "comparison" (see above).
- "operator2": `vars.get("$local.operator2")`: The relational operator as special character, mainly to be used in SQL statements, e.g. ">".

Here are examples showing how the filterConditionProcess can be designed:

Example of filterConditionProcess evaluating value selection via combo box

```
// the first part of the array returned by filterValuesProcess, e.g., a UID
var rawvalue = vars.get("$local.rawvalue");

// the relational operator coded as Integer number (non-unique!), e.g. "2"
var operator = vars.get("$local.operator");

// the relational operator as special character to be used in SQL statements, e.g. ">"
var operator2 = vars.get("$local.operator2");

// the relational operator as cleartext in String format, e.g. "NOT_EQUAL"
var comparison = vars.get("$local.comparison");

// useful logging for understanding the above variables
// -> just try various values and relational operators
//    and inspect the log output
logging.log("--------------------> rawvalue = " + rawvalue);
logging.log("--------------------> operator = " + operator);
logging.log("--------------------> operator2 = " + operator2);
logging.log("--------------------> comparison = " + comparison);

// Example:
// Assuming that ANOTHERTABLE has been used in filterValuesProcess,
// so ANOTHERTABLE.ANOTHERTABLEID is here given in rawvalue
// and now used for filtering MYTABLE via its column ANOTHERTABLE_ID

var myPrimaryId = rawvalue;
var sqlCondition = "";

switch(comparison) {
    case "EQUAL":
        sqlCondition = newWhere("MYTABLE.ANOTHERTABLE_ID", myPrimaryId, SqlBuilder.EQUAL());
        break;
    case "NOT_EQUAL":
        sqlCondition = newWhere("MYTABLE.ANOTHERTABLE_ID", myPrimaryId, SqlBuilder.NOT_EQUAL());
        break;
    case "ISNULL":
        sqlCondition = "MYTABLE.ANOTHERTABLE_ID IS NULL";
        break;
    case "ISNOTNULL":
        sqlCondition = "MYTABLE.ANOTHERTABLE_ID IS NOT NULL";
        break;
    default:
        sqlCondition = "1 = 2";
}

result.string(sqlCondition);
```

Example of filterConditionProcess evaluating value input via free text (no filterValuesProcess required in this case)

```
var myUserInput = vars.get("$local.rawvalue");

// operator selection is ignored here

var myFilterCondition = newWhere(
  "MYTABLE.MYCOLUMN",
  myUserInput,
  SqlBuilder.EQUAL());

result.string(myFilterCondition);
```

The specific code of the filterConditionProcess suitable for the carpool example will be added in a future version of this manual. You may try it by yourself meanwhile.

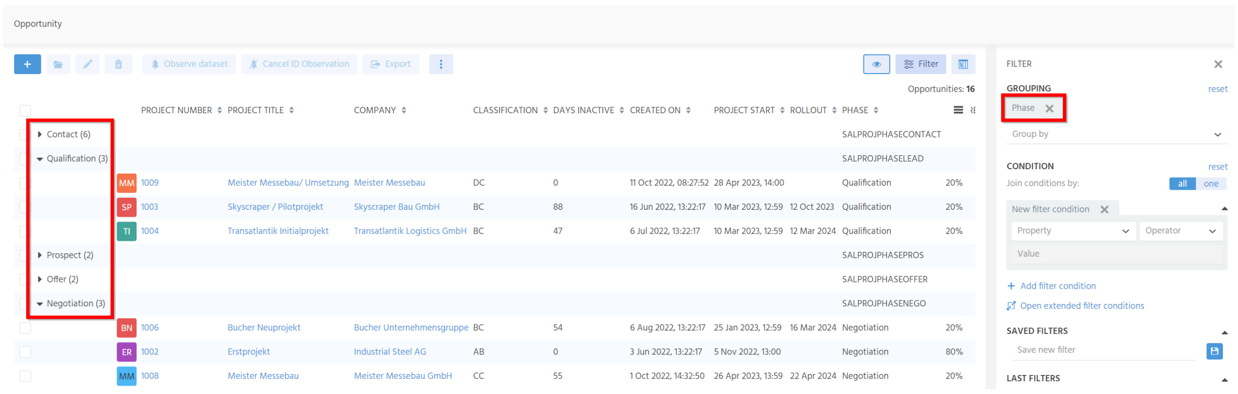
10.9.3.1.6. groupQueryProcess

Property groupQueryProcess is an option to group data provided via a FilterExtension. It requires no specific EntityField, because the grouping is created in the process itself. The result of the groupQueryProcess is an SQL string that returns the grouping.

The groupQueryProcess is triggered when the client user selects a FilterExtension-related "Group-by" value in the "Grouping" section of the filter component of the FilterView.

The grouping will only work if a filterConditionProcess is defined.

Example in xRM:
groupQueryProcess of Phase_filter ("Phase_filterExtention"), a FilterExtension of the "db" RecordContainer of Salesproject_entity ("Opportunity").


Opportunity

Observe dataset · Cancel ID Observation · Export · Filter

Opportunities: 16

		PROJECT NUMBER	PROJECT TITLE	COMPANY	CLASSIFICATION	DAYS INACTIVE	CREATED ON	PROJECT START	ROLLOUT	PHASE	
Contact (6)										SALPROJPHASECONTACT	
Qualification (3)										SALPROJPHASELEAD	
	MM	1009	Meister Messebau/ Umsetzung	Meister Messebau	DC	0	11 Oct 2022, 08:27:52	28 Apr 2023, 14:00		Qualification	20%
	SP	1003	Skyscraper / Pilotprojekt	Skyscraper Bau GmbH	BC	88	16 Jun 2022, 13:22:17	10 Mar 2023, 12:59	12 Oct 2023	Qualification	20%
	TI	1004	Transatlantik Initialprojekt	Transatlantik Logistics GmbH	BC	47	6 Jul 2022, 13:22:17	10 Mar 2023, 12:59	12 Mar 2024	Qualification	20%
Prospect (2)										SALPROJPHASEPROS	
Offer (2)										SALPROJPHASEOFFER	
Negotiation (3)										SALPROJPHASENEGO	
	BN	1006	Bucher Neuprojekt	Bucher Unternehmensgruppe	BC	54	6 Aug 2022, 13:22:17	25 Jan 2023, 12:59	16 Mar 2024	Negotiation	20%
	ER	1002	Erstprojekt	Industrial Steel AG	AB	0	3 Jun 2022, 13:22:17	5 Nov 2022, 13:00		Negotiation	80%
	MM	1008	Meister Messebau	Meister Messebau GmbH	CC	55	1 Oct 2022, 14:32:50	26 Apr 2023, 13:59	22 Apr 2024	Negotiation	20%

FILTER

GROUPING — reset

Phase ✕

Group by

CONDITION — reset

Join conditions by: [✓ all] one

New filter condition ✕

Property · Operator · Value

+ Add filter condition

Open extended filter conditions

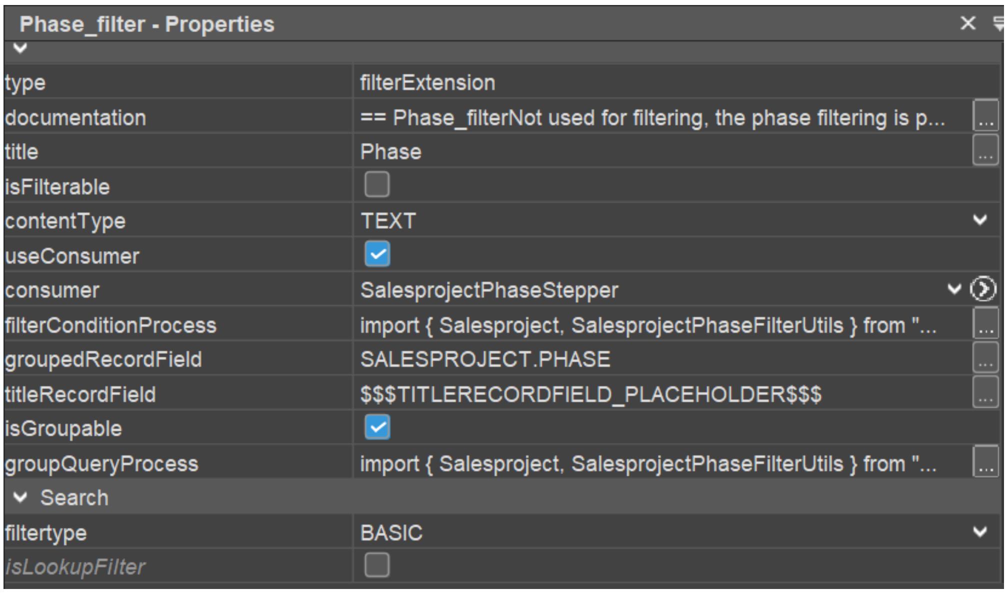
SAVED FILTERS

Save new filter

LAST FILTERS

As you can see, this FilterExtension is not used for filtering (which you can additionaly do via the EntityField PHASE). Its purpose is to enable grouping of Opportunity datasets by PHASE and still having them in the correct alphabetical order.

This is the configuration of Phase_filter:


Phase_filter - Properties

type	filterExtension
documentation	== Phase_filterNot used for filtering, the phase filtering is p...
title	Phase
isFilterable	☐
contentType	TEXT
useConsumer	✓
consumer	SalesprojectPhaseStepper
filterConditionProcess	import { Salesproject, SalesprojectPhaseFilterUtils } from "...
groupedRecordField	SALESPROJECT.PHASE
titleRecordField	$$$TITLERECORDFIELD_PLACEHOLDER$$$
isGroupable	✓
groupQueryProcess	import { Salesproject, SalesprojectPhaseFilterUtils } from "...
Search	
filtertype	BASIC
isLookupFilter	☐

Step-by-step explanation of how to implement a groupQueryProcess:

- Add a new FilterExtension.
- Check property "isGroupable" (otherwise, the grouping option is not visible in the client)
- Set groupedRecordField: This is the EntityField by which the grouping will be done.
- Set titleRecordField: This is the EntityField that will later be used as displayValue for the groups. As in this property a string can be entered, you can alternatively set a placeholder string and replace it later, e.g., by a join or (if it is not too long) subselect/caseWhen statement - see groupQueryProcess of Phase_filter.
- groupQueryProcess: Here, you are free to do what is required, as long as the result is a suitable

SQL string that includes a "group by" clause. You may have a look at the groupQueryProcess of Phase_filter to learn the approach. Here you have access to various useful variables (see chapter FilterExtensionSet). Variables used in Phase_filter are:

  - $local.condition: The condition that is given by filter and filterConditionProcess. If present, the condition needs to be appended to the SQL (see Phase_filter)
  - $local.count: Boolean indicating if the process is executed to calculate the count or for loading the data itself. If only the count is needed, the SQL should, for performance reasons, select only something like "1". If the data is to be loaded, take the column list (see below) and replace the placeholder text for the displayValue (if required, see Phase_filter)
  - $local.columnlist: string with columns, separated by comma (order: groupedRecordField, titleRecordField [, n aggregate fields])

Filter extensions are not automatically respected by the index. This means, if, e.g., you want to use a FilterExtension like "Supervisor assignment equals YES" in the access rights, you need to re-build this filter in the index. If you do not do this, then, in the index, there will be shown no result for the respectiv IndexGroup. Find more information in the ADITO document AID093_Indexsearch.pdf.

10.9.3.1.7. supportsFilterExtensionGrouping

It is possible to use FilterExtensions and FilterExtensionSets (see chapter FilterExtensionSet) on groups with RecordContainers without paging. If the property "isPageable" is disabled on a RecordContainer, the additional property `supportsFilterExtensionGrouping` is shown. If this property is set to `true`, the FilterExtensions are also shown when using grouping.

This has to be used with care, because it leads to all data being reloaded for **every** row of grouped data. The mechanism may cause a huge amount of data being loaded and most likely will negatively affect the system's performance!

10.9.3.2. Specific example task

In Context CarDriver, a filter option should be added that enables us to see only drivers who have ever reserved a specific car. In terms of the FilterView,

- the filter's "Property" is "has reserved car"
- the filter's "Operator" is "equal"
- the filter's "Value" is a list showing all cars

This feature can be generated as follows:

10.9.3.2.1. Creating a new FilterExtension

Open CarDriver_entity in the Navigator, right-click on its RecordContainer, and choose "Add Filter Extension" from the context menu. Enter a name of your choice, e.g., "hasReservedCar".

10.9.3.2.2. Setting the FilterExtension's properties

Fill the new FilterExtension's properties as follows:

======= General properties

- "title": Enter the text of the respective list item to appear in the filter's combo box "Property", e.g. "Has reserved car".
- "contentType": Enter the data type of the values to be entered or selected in the filter's field "Value". This data type will then, amongst others, determine the list of relational operators from which the user can select in the filter component's field "Operator". For our example, the contentType should be set to "TEXT".

======= Further properties

Now, try to complete the example task on your own, by configuring all required further properties/processes.

A sample solution will be added in a future version of this manual.

10.9.4. FilterExtensionSet

You will understand the content of this chapter better, if you first read the previous chapter FilterExtension.

FilterExtensionSet is an ADITO model used for extending the standard filter (controlled via the "isFilterable" properties of the record fields in the RecordContainer) by additional filter options, partly similar to the FilterExtension, but more complex and powerful.

A FilterExtensionSet can be generated as follows:

Open an Entity in the Navigator, right-click on its RecordContainer, and choose "Add Filter Extension Set" from the context menu. Enter a name of your choice.

10.9.4.1. Example

Here is an example of how to configure a FilterExtensionSet: Given we want to manage trainees, including their performance at school (grades in English, German, and math).

This example of a FilterExtensionSet is to demonstrate the 3 options to load the values (directly from the database; dropdown with filter values from filterValuesProcess; dropdown with filter values from Consumer) and the grouping.

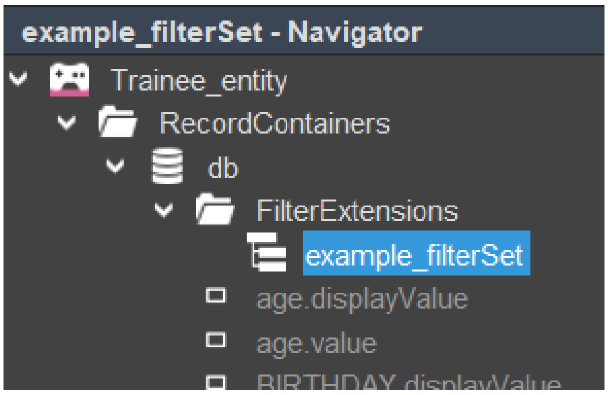
In the filterConditionProcess, in turn, we again have 3 options that are quite common with FilterExtensions: A boolean evaluation (yes/no), and two evaluations with type TEXT (one of it simple, and one more complex).

All defined filter and groupings in the set are directly refering to the same table as the RecordContainer does - therefore, the examples are a little bit "artificial", but nevertheless comparably easy to understand, as you do not have to deal with subqueries etc.

This example, for itself, is not meant as "best practice", but it demonstrates well how to handle a FilterExtensionSet, without having the need to call complex functions or generate/require complex SQL queries.

Now, first, we set up a Context "Trainee", including a "Trainee_entity" with several EntityFields. And we include this Context in a new menu group in the Global Menu. Furthermore, we create a database table and connect it with the Entity and its EntityFields. Most of this preparatory work can be realized via Liquibase: Please update your ADITO project, including its databas, using the corresponding Liquibase and .aod files in appendix Trainee example. Then, everything will be prepared to continue with the following paragraphs.

In the project tree, double-click on Trainee_entity and unfold its RecordContainer "db" in the Navigator window. There, if you unfold the node "FilterExtensions", you see a FilterExtensionSet named "example_filterSet":



Now we will configure this FilterExtensionSet’s properties step-by-step, along with some explanations as code comments. Furthermore, reading the various properties' property description will help you to understand the example.

First, make sure that property "filtertype" is set to BASIC. (The other option, "EXTENDED", would mean that the FilterExtensionSet’s features are only available via "Open extended filter conditions".)

10.9.4.1.1. Creating Consumer for gender-related field

The following code snippet is only required for making the example work properly. It is not directly related to the basics of a FilterExtensionSet.

The code of the valueProcess of ContainerName_param of Consumer "KeywordGenders"

```
import { result } from "@aditosoftware/jdito-types";
import { $SalutationKeywords } from "SalutationKeywords_registry";

result.string($SalutationKeywords.personGender());
```

10.9.4.1.2. filterFieldsProcess

The code of "example_filterSet"'s property filterFieldsProcess

```
import { result } from "@aditosoftware/jdito-types";
import { KeywordUtils } from "KeywordUtils_lib";
import { $SalutationKeywords } from "SalutationKeywords_registry";

//no local variables available

var filterFields = [
    //No dropdown
    {
        name: "FILTER_GRADEENGLISH",
        title: "Grade English entered?",
        contentType: "BOOLEAN",
        isGroupable: true,

        groupedRecordField:"CASE WHEN TRAINEE.GRADEENGLISH IS NOT NULL THEN 1 ELSE 0 END",
        titleRecordField:"CASE WHEN ISNULL(TRAINEE.GRADEENGLISH) = 0 THEN 'Ja' ELSE 'Nein' END",
    },
    {
        name: "FILTER_GRADEGERMAN",
        title: "Grade German entered?",
        contentType: "BOOLEAN",
        isGroupable: true,

                groupedRecordField:"CASE WHEN TRAINEE.GRADEGERMAN IS NOT NULL THEN 1 ELSE 0 END",
        titleRecordField:"CASE WHEN ISNULL(TRAINEE.GRADEGERMAN) = 0 THEN 'Ja' ELSE 'Nein' END",
    },
    {
        name: "FILTER_GRADEMATH",
        title: "Grade math entered?",
        contentType: "BOOLEAN",
        isGroupable: true,

                groupedRecordField:"CASE WHEN TRAINEE.GRADEMATH IS NOT NULL THEN 1 ELSE 0 END",
        titleRecordField:"CASE WHEN ISNULL(TRAINEE.GRADEMATH) = 0 THEN 'Ja' ELSE 'Nein' END",
    },

    //dropdown => uses filterValuesProcess (only for this one)
    {
        name: "FILTER_GRADE",
        title: "Grade",
        contentType: "TEXT",
        hasDropDownValues: true,
        isGroupable: false,
    },

    //dropdown => uses Consumer in current entity
    {
        name: "FILTER_GENDER",
        title: "Gender",
        contentType: "TEXT",
        hasDropDownValues: true,
        isGroupable: true,
        consumer: "KeywordGenders",

        groupedRecordField:"TRAINEE.GENDER",
        titleRecordField:KeywordUtils.getResolvedTitleSqlPart($SalutationKeywords.personGender(), "TRAINEE.GENDER")
    }

];

result.string(JSON.stringify(filterFields));
```

10.9.4.1.3. filterValuesProcess

The code of "example_filterSet"'s property filterValuesProcess

```
import { logging, result, vars } from "@aditosoftware/jdito-types";

let filter = JSON.parse(vars.getString("$local.filter"));

let values = [];
switch(filter.name){
    case "FILTER_GRADE":
        values = [
            ["5", "excellent"],
            ["4", "good"],
            ["3", "satisfactory"],
            ["2", "less than satisfactory"],
            ["1", "unsatisfactory"]
        ]
        break;
}

result.object(values);
```

10.9.4.1.4. filterConditionProcess

The code of "example_filterSet"'s property filterConditionProcess

```
import { logging, result, vars } from "@aditosoftware/jdito-types";
import { newWhere, SqlBuilder } from "SqlBuilder_lib";

//all possible local variables
//let columnPlaceholder = vars.get("$local.columnPlaceholder");
//let columntype = vars.get("$local.columntype");
//var comparison = vars.get("$local.comparison");
//let condition = vars.get("$local.condition");
//let conditionHaving = vars.get("$local.conditionHaving");
//let isAggregateCondition = vars.get("$local.isAggregateCondition");
//let name = vars.get("$local.name");
//let operator = vars.get("$local.operator");
//let operator2 = vars.get("$local.operator2");
//let placeholder = vars.get("$local.placeholder");
//let rawvalue = vars.get("$local.rawvalue");
//let value = vars.get("$local.value");

let rawValue = vars.get("$local.rawvalue");
let comparison = vars.get("$local.comparison");

let name = vars.get("$local.name"); //e.g. Trainee_entity.example_filterSet.FILTER_GRADEMATH
let filterName = name.split(".")
    .pop(); // e.g. FILTER_GRADEMATH
let column = "TRAINEE." + name.split("_")
    .pop(); //e.g. GRADEMATH

let cond = newWhere();

switch (filterName)
{
    case "FILTER_GRADEENGLISH":
    case "FILTER_GRADEGERMAN":
    case "FILTER_GRADEMATH":
        {
            let nullOperator = "IS NULL";
            switch (comparison)
            {
                case "EQUAL":
                    nullOperator = rawValue == 1 ? "IS NOT NULL" : "IS NULL";
                    break;
                case "NOT_EQUAL":
                    nullOperator = rawValue == 1 ? "IS NULL" : "IS NOT NULL";
                    break;
                case "ISNULL":
                    nullOperator = "IS NULL";
                    break;
                case "ISNOTNULL":
                    nullOperator = "IS NOT NULL";
                    break;
            }

            cond.and(column + " " + nullOperator);
        }
        break;
    case "FILTER_GRADE":
        {
            let operator = null;

            switch (comparison)
            {
                case "EQUAL":
                    cond.and(
                        newWhere("TRAINEE.GRADEENGLISH", rawValue, SqlBuilder.EQUAL())
                            .or("TRAINEE.GRADEGERMAN", rawValue, SqlBuilder.EQUAL())
                            .or("TRAINEE.GRADEMATH", rawValue, SqlBuilder.EQUAL())
                    );
                    break;
                case "NOT_EQUAL":
                    cond.and(
                        newWhere("TRAINEE.GRADEENGLISH", rawValue, SqlBuilder.NOT_EQUAL())
                            .or("TRAINEE.GRADEGERMAN", rawValue, SqlBuilder.NOT_EQUAL())
                            .or("TRAINEE.GRADEMATH", rawValue, SqlBuilder.NOT_EQUAL())
                    );
                    break;
                case "ISNULL":
                    cond.and(
                        newWhere("TRAINEE.GRADEENGLISH IS NULL")
                            .and("TRAINEE.GRADEGERMAN IS NULL")
                            .and("TRAINEE.GRADEMATH IS NULL")
                    );
                    break;
                case "ISNOTNULL":
                    cond.and(
                        newWhere("TRAINEE.GRADEENGLISH IS NOT NULL")
                            .and("TRAINEE.GRADEGERMAN IS NOT NULL")
                            .and("TRAINEE.GRADEMATH IS NOT NULL")
                    );
                    break;
            }
        }
        break;

    case "FILTER_GENDER":
        {
            let operator = null;

            switch (comparison)
            {
                case "EQUAL":
                    cond.and(column, rawValue, SqlBuilder.EQUAL());
                    break;
                case "NOT_EQUAL":
                    cond.and(column, rawValue, SqlBuilder.NOT_EQUAL());
                    break;
                case "ISNULL":
                    cond.and(column + " IS NULL");
                    break;
                case "ISNOTNULL":
                    cond.and(column + " IS NOT NULL");
                    break;
            }
        }
        break;
}

logging.log(JSON.stringify({
    rawValue,
    comparison,
    name,
    cond: cond.toString()
}, null, "\t"));

result.string(cond.toString());
```

10.9.4.1.5. groupQueryProcess

The code of "example_filterSet"'s property groupQueryProcess

```
import { logging, result, vars } from "@aditosoftware/jdito-types";
import { newWhere, SqlBuilder } from "SqlBuilder_lib";

//all possible local variables
//let columnlist = vars.get("$local.columnlist");
//let columns = vars.get("$local.columns");
//let columntype = vars.get("$local.columntype");
//let condition = vars.get("$local.condition");
//let contenttype = vars.get("$local.contenttype");
//let count = vars.get("$local.count");
//let fieldname = vars.get("$local.fieldname");
//let grouped = vars.get("$local.grouped");
//let groupedlist = vars.get("$local.groupedlist");
//let name = vars.get("$local.name");
//let order = vars.get("$local.order");

var sql = new SqlBuilder()

if (vars.get("$local.count")) // TRUE if the count of the records is needed
{
    sql.select("1");
}
else
{
    let columnlist = vars.get("$local.columnlist");

    sql.select([columnlist]);
}

sql.from("TRAINEE");

let condition = vars.get("$local.condition");
if(condition != "  ")
{
    sql.where(condition);
}

let grouped = vars.get("$local.grouped");
sql.groupBy(grouped);

sql.orderBy(grouped);

result.string(sql.toString());
```

10.9.4.2. Further examples

Further, more complex examples of FilterExtensionSets are included in the ADITO xRM project. For example, a FilterExtensionSet named "Attribute_filter" is included in the RecordContainer of several Entities, e.g., in Organisation_entity and in Person_entity. This FilterExtensionSet enables the client user to filter the Entity’s datasets according to the attributes assigned to them (e.g., the attribute "Loyalty" of Organisation/Company or Person/Contact datasets).

10.9.4.3. Available local variables

The following "$local" variables can, amongst others, be accessed in the code of a FilterExtensionSet’s properties:

Name	Description
$local.count	TRUE if the count of the records is needed
$local.columnlist	String with the columns (and expressions) expected to be returned by the query
$local.condition	the (filter) condition that’s being used (if used in a grouping, then it includes the group hierarchy); see example in appendix ("$local variables")
$local.groupedlist	String with the columns (and expression) used for grouping
$local.order	String that contains the order expression how the grouped items have to be sorted
$local.name	String value of the "name" property, if a filterField was returned by the filterFieldsProcess; every filterField has its unique name

10.9.4.4. useConsumer

If you are already familiar with FilterExtensionSets, please note that the "useConsumer" functionality (see chapter "FilterExtension" above) is also available for FilterExtensionSets. You can configure it via the JSON config object that is set in the result of the filterFieldsProcess: Simply add attribute "consumer" and set the Consumer’s name as its argument. Here is a universal code example:

Example of result of filterFieldsProcess relating to a Consumer

```
var myConfig = [];
(...)
    myConfig.push({
        name: (...),
        title: (...),
        contentType: (...),
        hasDropDownValues:  (...)
        isGroupable:  (...),
        groupedRecordField:  (...),
        titleRecordField:  (...),
        consumer: "MyConsumerName",
        (...)
    });

myConfig = JSON.stringify(myConfig);
result.string(myConfig);
```

10.9.4.5. groupQueryProcess

Grouping via groupQueryProcess is also available for FilterExtesionSets. The approach is similar to the groupQueryProcess of FilterExtensions. The difference is only that the required properties "groupedRecordField" and "titleRecordField" are filled in the filterFieldsProcess. In the groupQueryProcess the usual $local variables are available ($local.columnlist, $local.condition, $local.groupedlist, etc.), and you can use them to build your SQL statement that makes the grouping.

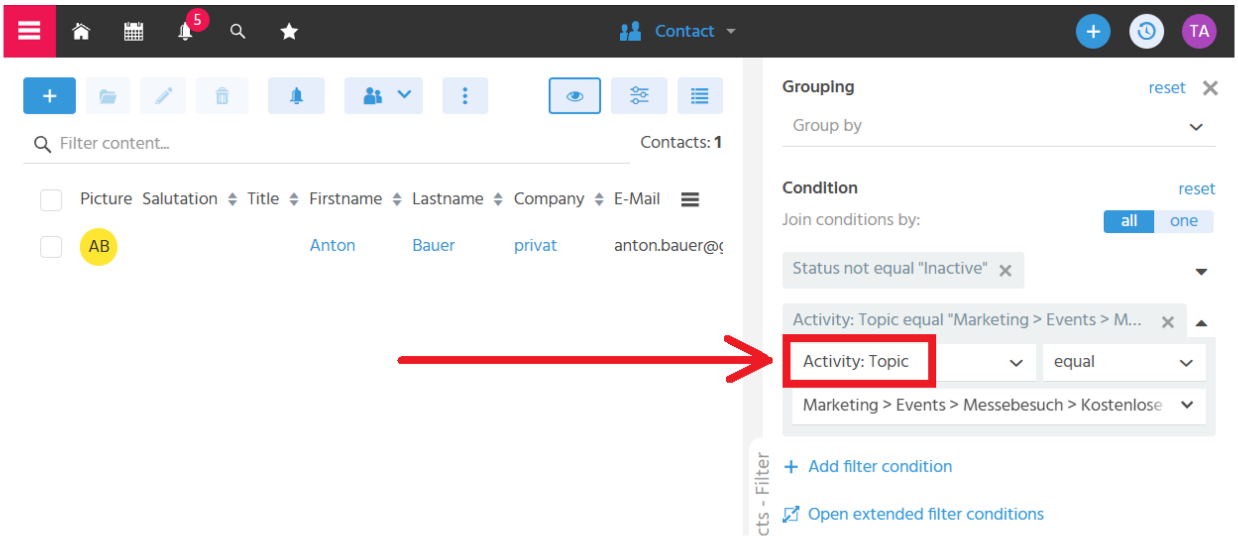
You may study an example in xRM, e.g., groupQueryProcess of ClassificationGroup_filter, a FilterExtensionSet of several Entities, e.g., of Organisation_entity or Salesproject_entity.

10.9.5. Consumer filter

A Consumer filter makes filter criteria of a dependent Entity available. This enhances the range of filtering options.

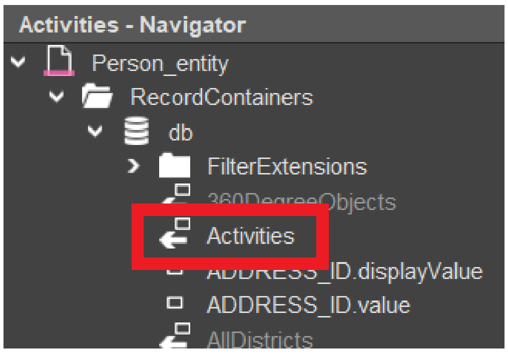
Example included in the ADITO xRM project:

In Context "Person" (titled "Contact" in the Global Menu), you have the option to filter contact persons according to the topic of a connected Activity:



This was made possible by the following steps:

- In the RecordContainer, the ConsumerMapping for Consumer "Activities" was initialized. ("Click to initialize" in the property sheet)



- In the property sheet of this ConsumerMapping, the following properties were set:
  - filtertype: EXTENDED
  - isFilterable: true
  - filterConditionProcess:

```
import { result, vars  } from "@aditosoftware/jdito-types";
import { newSelect, newWhere, SqlBuilder } from "SqlBuilder_lib";

let subselect = newSelect("CONTACT.PERSON_ID")
    .from("CONTACT")
    .join("ACTIVITYLINK", "ACTIVITYLINK.OBJECT_ROWID = CONTACT.CONTACTID")
    .and("ACTIVITYLINK.OBJECT_TYPE", "Person")
    .join("ACTIVITY", "ACTIVITY.ACTIVITYID = ACTIVITYLINK.ACTIVITY_ID")
    .where(vars.get("$local.condition"));

let sql = newWhere("PERSONID", subselect, SqlBuilder.IN());

result.string(sql.toString());
```

10.9.6. EntityRecordsRecipe

As the name suggests, EntityRecordsRecipe is a definition ("recipe") of datasets (records) of a specific Entity. It can be built and configured quite intuitively, with several options, e.g., to specify a filter (via a FilterGroup object, see chapter FilterBuilder) or a list (as array) of UIDs of records to be excluded.

You can consider EntityRecordsRecipe to be a kind of extended filter that can be applied, e.g., when a Context is opened, or when records are loaded via LoadEntity.

10.9.6.1. Technical background

An EntityRecordsRecipe does NOT hold the records (datasets) theirselves, but it *defines* them - like a recipe defines the ingredients required to cook a meal. Basically, it’s a filter, whose result can optionally be further reduced by a set of "UIDs to be excluded".

Formerly, this definition could exclusively be done via a set of the single UIDs of all corresponding records. This principle

- did not scale very well, as additional or changed records always meant that the set of UIDs had to be adjusted-
- could cause performance and memory issues, if the Entity held millions of records, causing a huge set of UIDs.

Therefore, EntityRecordsRecipe was introduced, providing a scalable and well-performing solution for defining an entirety of records without having to indicate every single UID. An example in the client showing the benefit of this approach is the "select all" button of a table: If you check it and subsequently uncheck 3 single records, an EntityRecordsRecipe is being built defining "all datasets without UIDs xxx, yyy, and zzz". (This EntityRecordsRecipe is available in variable "$sys.selectionsRecordsRecipe" - see chapter further below.)

10.9.6.2. General usage

The ADITO system uses EntityRecordsRecipe internally, e.g., to hold and process a definition of the records selected by the client user - see chapter on variable "$sys.selectionsRecordsRecipe" further below.

If you, however, want to create a new EntityRecordsRecipe, then proceed as follows.

The first step is to create a builder object:

```
var entityRecordsRecipeBuilder = neonFilter.createEntityRecordsRecipeBuilder();
```

This object has several methods, allowing to specify various options. Each method returns the builder object itself, so a "chaining" of methods is possible - similar to LoadEntity or "SqlBuilder". Technically, the conditions resulting from these methods are combined with a logical "AND".

The most common methods to call are:

- `entity(<Name of Entity>)`: Definition of the Entity, given as its name String, e.g., `.entity("Person_entity")`. If this is the only method you call, then the definition is "ALL records of the Entity". In SQL terms, the effect of this method is "… FROM <joined database tables of the Entity’s RecordContainer>".
- `filter(<filter>)`: Filter that restricts the record definition to specific conditions. The filter can be given as FilterGroup object (see chapter FilterBuilder) or as JSON String. In SQL terms, the effect of this method is "… WHERE <filter conditions>".
- `.uidsExcludelist(<Array of UIDs>)`: Specifies a list of UIDs of all records that are to be excluded from the remaining records. In SQL terms, the effect of this method is "… WHERE xxxID NOT IN ('UID1', 'UID2', …)".
- `.uidsIncludelist(<Array of UIDs>)`: Specifies a list of UIDs of records to which the definition is to be restricted. In other words: This is the "maximum" of records - which can be further reduced by the conditions given in the other methods (`.uidsExcludelist` or `.filter`). Do not be mislead by the name: This list does NOT "add" additional records to those specified in `.filter`, but it defines that the UIDs of the records described by the EntityRecordsRecipe must be included in this list of UIDs. Or, in SQL terms, the effect of this method is a condition like "… where MYTABLEID in ('UID1', 'UID2', …)".
  NOTE: If the argument of method `uidsIncludelist` is
  - an empty Array, then *nothing* is loaded subsequently
  - null, then there there is *not any UID-related restriction at all* (= same as if this setter method was not executed at all)

10.9.6.3. Usage in "openContextWithRecipe"

The following example will help you to understand how the EntityRecordsRecipe works. This code opens the FilterView of Person_entity, with the datasets

- restricted to persons whose last name starts with letter "B"
- without (excluding) the persons "Frank Baer" and "Christine Burger"

You may include this code, e.g., in the onActionProcess of a test Action of Person_entity. (see chapter Actions)

Person_entity.TestActionGroup.testAction.onActionProcess

```
// Definition of test filter that is restricting records
// to those with LASTNAME starting with letter "B"
var myFilterCondition = neonFilter.createFilterCondition()
.field("LASTNAME")
.searchOperator(neonFilter.SEARCH_OPERATOR_STARTSWITH)
.contentType(neonFilter.CONTENT_TYPE_TEXT)
.key("B");
var myFilter = neonFilter.createFilterGroup()
.addFilterCondition(myFilterCondition);

// Definition of array holding UIDs (CONTACTIDs)
// of "Frank Baer" and "Christine Burger"
var myUidList = ["701569b7-d791-4682-89a1-bf26682187af", "a38a19f6-6255-47b0-bbea-138bae2271c4"];

// Definition of EntityRecordsRecipeBuilder
var myEntityRecordsRecipe = neonFilter.createEntityRecordsRecipeBuilder()
.entity("Person_entity")
// applying filter
.filter(myFilter)
// excluding records of "Frank Baer" and "Christine Burger"
.uidsExcludelist(myUidList);

// opens PersonFilter_view with 1 record ("Carl Bush")
neon.openContextWithRecipe("Person", "PersonFilter_view", myEntityRecordsRecipe,
    neon.OPERATINGSTATE_SEARCH, null, false);
```

Here is a modified example, without using method `filter`, resulting in exactly 2 records ("Frank Baer" and "Christine Burger"):

Person_entity.TestActionGroup.testAction2.onActionProcess

```
(...)
var myEntityRecordsRecipe = neonFilter.createEntityRecordsRecipeBuilder()
.entity("Person_entity")
// restricting to records of "Frank Baer" and "Christine Burger"
.uidsIncludelist(myUidList);
```

And here is an example that combines a filter with a "include list" (resulting in only 1 record, "Christine Burger"):

Person_entity.TestActionGroup.testAction3.onActionProcess

```
// Definition of test filter that is restricting records
// to those with LASTNAME starting with letter "B"
var myFilterCondition = neonFilter.createFilterCondition()
.field("FIRSTNAME")
.searchOperator(neonFilter.SEARCH_OPERATOR_STARTSWITH)
.contentType(neonFilter.CONTENT_TYPE_TEXT)
.key("C");
var myFilter = neonFilter.createFilterGroup()
.addFilterCondition(myFilterCondition);

// Definition of array holding UIDs (CONTACTIDs)
// of "Frank Baer" and "Christine Burger"
```

```
var myUidList = ["701569b7-d791-4682-89a1-bf26682187af", "a38a19f6-6255-47b0-bbea-138bae2271c4"];

// Definition of EntityRecordsRecipeBuilder
var myEntityRecordsRecipe = neonFilter.createEntityRecordsRecipeBuilder()
.entity("Person_entity")
// applying filter
.filter(myFilter)
// restricting to records of "Frank Baer" and "Christine Burger"
.uidsIncludelist(myUidList);

// opens PersonFilter_view with 1 record ("Christine Burger")
neon.openContextWithRecipe("Person", "PersonFilter_view", myEntityRecordsRecipe,
    neon.OPERATINGSTATE_SEARCH, null, false);
```

10.9.6.4. Usage in "LoadEntity"

EntityRecordsRecipe can also be used to define the records to load via "LoadEntity" (see appendix LoadEntity): Simply specify an EntityRecordsRecipe instance as parameter of the LoadRowsConfig’s method `fromEntityRecordsRecipe`. Here is an example using the same EntityRecordsRecipe as in the example code of the previous chapter:

Person_entity.TestActionGroup.testAction4.onActionProcess

```
(...)
var myEntityRecordsRecipe = (...) // see previous chapter

var myLoadRowsConfig = entities.createConfigForLoadingRows()
// can be skipped, as it is included in EntityRecordsRecipe
//.entity("Person_entity")
.fields(["FIRSTNAME", "LASTNAME"])
.fromEntityRecordsRecipe(myEntityRecordsRecipe)

var myRows = entities.getRows(myLoadRowsConfig);

// [{"FIRSTNAME":"Carl", "LASTNAME":"Bush"}]
logging.log(JSON.stringify(myRows));
```

10.9.6.5. Usage in customized methods

In principle, there are almost no limits for integrating EntityRecordsRecipe also in customized methods. In fact, your application’s performance can be significantly improved if you use it whenever required, or if you refactor your existing customized code with respect to EntityRecordsRecipe.

Here is an example of how an integration can look like in a customized code (just as pattern):

Example pattern for using EntityRecordsRecipe in a customized method

```
var attributeValue = (...);
var activeStatusFilter = neonFilter.createFilterCondition()
.field("STATUS")
.searchOperator(neonFilter.SEARCH_OPERATOR_EQUAL)
.key($KeywordRegistry.contactStatus$active())
.contentType(neonFilter.CONTENT_TYPE_TEXT);

var affectedContactsRecordsRecipe = neonFilter.createEntityRecordsRecipeBuilder()
.entity("Person_entity")
.filter(activeStatusFilter)
.parameters({"NoCommRestriction_param": "EMAIL"});

AttributeRelationUpdateUtils.addAttribute(
  $AttributeRegistry.deliveryTerm(),
  "Person",
  affectedContactsRecordsRecipe,
  attributeValue);
```

The argument of method `.filter` can either be a filter object (as shown in the above example), but it could also be a JSON filter object (stringified) instead. Method `.filter` only works, if also method `.entity` is used (unlike parameters, uids, etc.).

10.9.6.6. $sys.selectionsRecordsRecipe

The system variable "$sys.selectionsRecordsRecipe" holds an EntityRecordsRecipe describing the Entity’s name and the records selected by the client user.

The ADITO system automatically reacts to changes of the value of "$sys.selectionRecordsRecipe" - i.e., subsequent processes will be executed, e.g., the titleProcess of an Action.

The content of an EntityRecordsRecipe depends on the value of the Entity’s property "recordsRecipeSupported". If this property’s value is

- false, then the description of the selected records is exclusively done via "uidsIncludelist" - while the variables "$sys.selections" and "$sys.selectionRows" are still working. This enables you to write and apply processes already using EntityRecordsRecipe even for Entitys that do not yet support it.
- true, then also EntityRecordsRecipe’s methods "filter" and "uidsExcludelist" might be used - depending on the situation ("select all" button checked or not, filter defined or not, etc.). For most Entitys, this state should be the standard.

If property recordsRecipeSupported is set to true, then variables "$sys.selections" and "$sys.selectionRows" will be deactivated (holding "null"). This is intended, in order to draw your attention to those parts of your code that have not yet been transferred to support of EntityRecordsRecipe.

> Please refer to the corresponding chapter in the Update Manual, which is available in the customer area of the ADITO web site.

(The above principle is also used for "$field.<consumer>.selectionRecordsRecipe".)

This is an example code of a test Action of Person_entity, opening the FilterView again, restricted to the selected records (i.e., all records of the FilterView with checkboxes checked by the client user).

Person_entity.TestActionGroup.testAction5.onActionProcess

```
var myEntityRecordsRecipeAsJSON = vars.get("$sys.selectionsRecordsRecipe");

// logging the selected records
logging.log("------> " + myEntityRecordsRecipeAsJSON);

neon.openContextWithRecipe("Person", "PersonFilter_view", myEntityRecordsRecipeAsJSON,
    neon.OPERATINGSTATE_SEARCH, null, false);
```

Now, to be more exact, `$sys.selectionsRecordsRecipe` holds the EntityRecordsRecipe not as EntityRecordsRecipeBuilder object, but as JSON string.

You can simply convert this JSON string into an EntityRecordsRecipeBuilder object, by specifying it as parameter of the create method:

```
var myEntityRecordsRecipe = neonFilter.createEntityRecordsRecipeBuilder(vars.get("$sys.selectionsRecordsRecipe"));
(...)
```

Subsequently, you may modify this object (e.g., by calling methods `filter` or `uidsExcludelist`- see previous chapter) and use it for any purpose.

10.9.6.7. Example: Notifications

In the xRM project' Notification_entity, you can find an example of the usage of EntityRecordsRecipe. In the client, a "select all" button is available, allowing the user to select all Notification records (and, if required, unselect single records afterwards) - including those that will appear not before you scroll down or to the next "page".

After selecting Notification records, the user can change their state all at once. In the corresponding Actions' onActionProcesses the EntityRecordsRecipe approach is used. See also the functionality of Notification_lib and the ADITO platform methods notification.xxx, e.g.

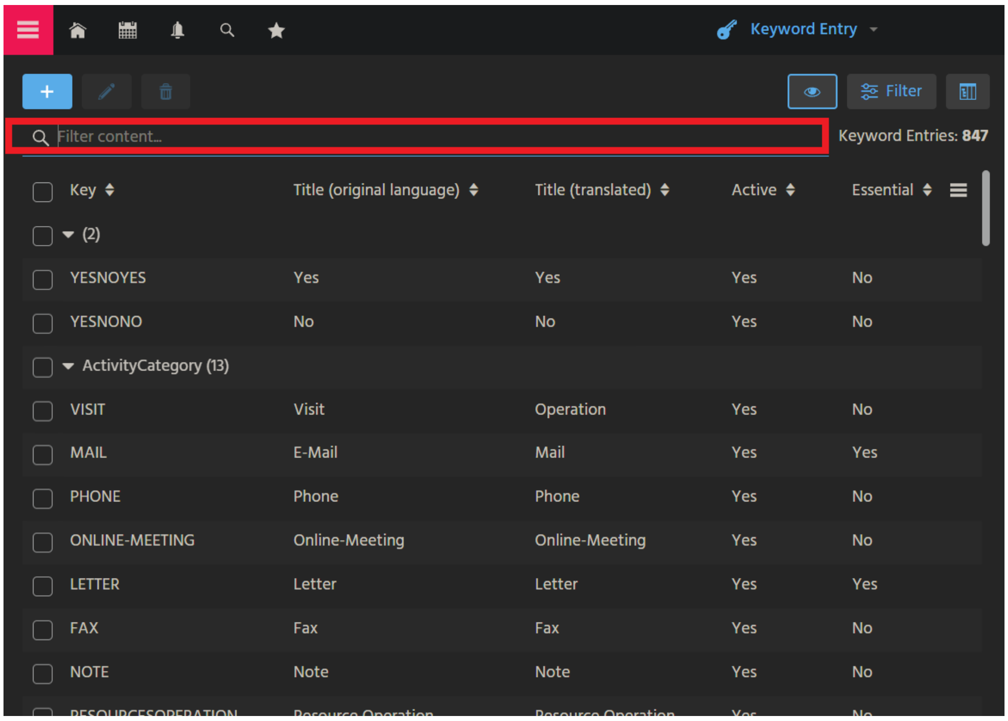
- `notification.updateUserNotificationsStateBulk`, which in turn requires method

- `notification.createUpdateStrategy`

10.9.7. Context filter (content search)

Several ViewTemplates provide the option to display a so-called Context filter. Its visual expression is the horizontal content search bar that is shown in the upper part of the ViewTemplate.

Example:


Keyword Entry

Filter

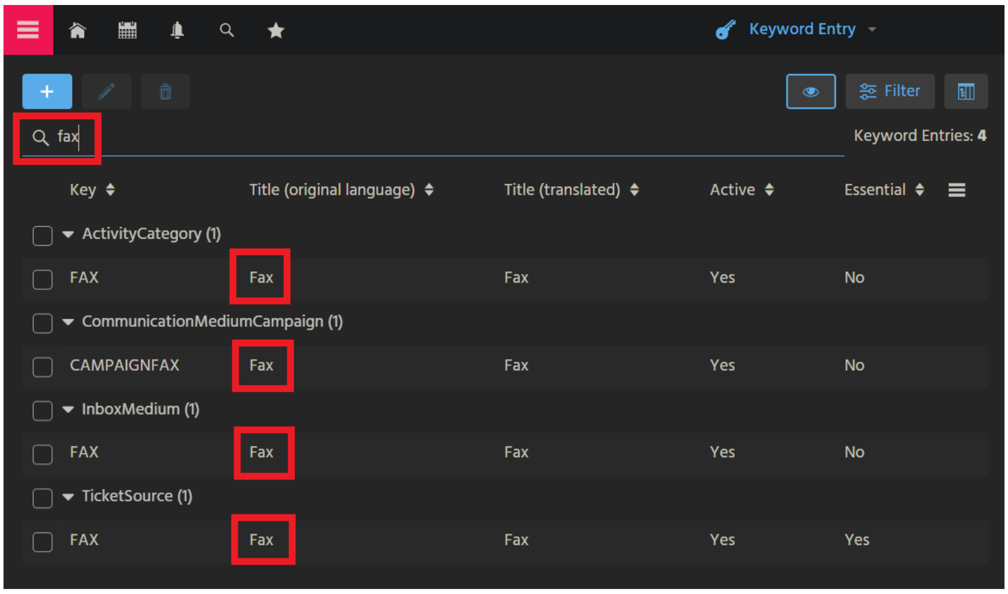
Filter content...

Keyword Entries: **847**

Key	Title (original language)	Title (translated)	Active	Essential
(2)				
YESNOYES	Yes	Yes	Yes	No
YESNONO	No	No	Yes	No
ActivityCategory (13)				
VISIT	Visit	Operation	Yes	No
MAIL	E-Mail	Mail	Yes	Yes
PHONE	Phone	Phone	Yes	No
ONLINE-MEETING	Online-Meeting	Online-Meeting	Yes	No
LETTER	Letter	Letter	Yes	Yes
FAX	Fax	Fax	Yes	No
NOTE	Note	Note	Yes	No

Figure 26. Example: The Context filter of Context KeywordEntry

→


Keyword Entry

Filter

fax

Keyword Entries: **4**

Key	Title (original language)	Title (translated)	Active	Essential
ActivityCategory (1)				
FAX	Fax	Fax	Yes	No
CommunicationMediumCampaign (1)				
CAMPAIGNFAX	Fax	Fax	Yes	No
InboxMedium (1)				
FAX	Fax	Fax	Yes	No
TicketSource (1)				
FAX	Fax	Fax	Yes	Yes

Figure 27. Example: Filter criteria "fax"

Via the Context filter, you can enter filter terms that are then processed in different ways, depending on the ViewTemplate's type and some specific settings. This chapter gives an overview and further details.

10.9.7.1. Availability

Specific ViewTemplate types are able to show and process a Context filter, if

1. their property `hideContentSearch` is set to false and
2. the corresponding RecordFieldMappings' (e.g., NAME.value) property `isLookupFilter` is set to true.

These are:

- BreadCrumbTreeTable: This type is not yet available via "Add View Template...", but it is automatically used on devices MOBILE and TABLET, if a Tree or TreeTable type is used, and property `useBreadCrumbs` is set to true.
- CardTable
- DynamicMultiDataChart
- DynamicSingleDataChart
- MultiEditTable
- ResourceTimeline
- Table
- Tiles
- Timeline
- Tree
- TreeTable

Furthermore, this filter/search feature is also available in the ADITO xRM project's View "DefaultLookup_view" (of Context "Default_context"), which has a ViewTemplate of type "List". Here, you can configure the lookup component that is to be used, if no specific lookupView is set in a Context. As the ListViewTemplate is not yet available via "Add View Template...", it needs to be configured in the source code of DefaultLookup_view:

Source code of "DefaultLookup_view", including its ListViewTemplate "DefaultList"

```
<?xml version="1.0" encoding="UTF-8"?>
<neonView xmlns="http://www.adito.de/2018/ao/Model" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" VERSION="1.2.3"
xsi:schemaLocation="http://www.adito.de/2018/ao/Model adito://models/xsd/neonView/1.2.3">
  <name>DefaultLookup_view</name>
  <majorModelMode>DISTRIBUTED</majorModelMode>
  <layout>
    <noneLayout />
  </layout>
  <children>
    <listViewTemplate>
      <name>DefaultList</name>
      <entityField>#ENTITY</entityField>
    </listViewTemplate>
  </children>
</neonView>
```

10.9.7.2. Evaluation

The term(s) entered in the Context filter are evaluated by different instances in different ways, depending on the ViewTemplate type and the RecordContainer’s property settings:

- The **ViewTemplate component** performs the filtering, if the RecordContainer’s properties `isPageable` and `isRequireContainerFiltering` are both set to false.
  - ResourceTimeline: Checks, if *at least one* of the strings of the filter value is included in the title (logical "OR" - unlike all other cases).
  - BreadCrumbTreeTable (see chapter Availability), Tiles, Tree, and TreeTable: Checks, if the columns of a row include *all* filter terms.
  - DynamicMultiDataChart and DynamicSingleDataChart: no filtering
- The **RecordContainer** performs the filtering, if it is filterable, i.e., if property `isRequireContainerFiltering` is set to true. Then, the filter checks, if the columns of a row include all filter terms. If, additionally, a filter is set in the FilterView’s filter component (right hand side of the FilterView), then a combined filter is constructed, joined with a logical "AND".
  - JDitoRecordContainer, IndexRecordContainer, DBRecordContainer: Filter will be set, and the data will be loaded anew.
  - DatalessRecordContainer or no RecordContainer: no filtering

10.10. RecordContainers

A RecordContainer is the ADITO model that defines the way how the data of an Entity is retrieved (loaded) and persisted (saved). There are various types of RecordContainers, which are described in the following chapters.

Depending on the purpose, it can be suitable to define more than one RecordContainer per Entity. An example of this is KeywordEntry_entity in the ADITO xRM project.

A RecordContainer includes the option to utilize a **cache**, in order to increase the performance of the ADITO system for repetitive requests of the same data. This is described in the appendix RecordContainerCache.

10.10.1. Database RecordContainer

A Database RecordContainer (dbRecordContainer) is a RecordContainer that enables an easy-to-establish connection of specific EntityFields with specific database columns. In the background, this RecordContainer automatically generates all required SQL statements for loading (SELECT), changing (UPDATE), saving (INSERT), ordering (ORDER BY), and deleting (DELETE) data.

Furthermore, you can optionally write specific parts of the SQL statement by yourself, e.g., by using

- the RecordContainer’s properties
  - fromClauseProcess (FROM)
  - conditionProcess (WHERE)
  - orderClauseProcess (ORDER BY)
- a RecordFieldMapping’s property "expression"

In the carpool example of this manual, you can find several examples of how to use a dbRecordContainer. Furthermore, you can find detailled information in appendix Database Access, chapter "Basic SQL Statement".

10.10.1.1. COUNT queries

When a dbRecordContainer loads data from the database, in many cases, a `SELECT COUNT(*)` statement is executed automatically, before the `SELECT` statement of the actual data is executed.

10.10.1.1.1. Purpose

The automatic `SELECT COUNT` queries have several reasons, particularly,

- the number of datasets is stored in specific variables, e.g., `$sys.datarowcount`
- the SELECT statement for retrieving the actual data is skipped, if `SELECT COUNT(*)` results in 0 (no datasets available).

10.10.1.1.2. minimizeCountQueries

Usually, a `SELECT COUNT(*)` statement consumes only minimal system resources. If, nevertheless, you want to reduce the frequency of `SELECT COUNT(*)` queries, you can set the dbRecordContainer’s property "minimizeCountQueries" to true. However, before using this property, read its property description carefully, in order to avoid unpleasant side-effects.

10.10.1.1.3. Caching not required

When your dbRecordContainer utilizes a cache, please note: Despite caching is active, still a `SELECT COUNT(*)` statement is executed. The reason for this is that `SELECT COUNT(*)` queries are generally excluded from being cached, as it is assumed that these queries consume only minimal system resources.

10.10.2. JDitoRecordContainer

10.10.2.1. Introduction

While a dbRecordContainer has a database as data source, a JDitoRecordContainer has JDito code as data source. (Of course, the JDito code itself can include a loading from the database.) The result of this code is an array. This array must provide the data in a specific order, which can be configured in the JDitoRecordContainer’s property "recordFieldMappings". The data source array is the result object of property "contentProcess".

To establish a JDitoRecordContainer, proceed as follows:

- Open the respective Entity in the Navigator window.
- Right-click on it and choose "New RecordContainer" from the context menu.
- A dialog appears, in which you choose "JDitoRecordContainer" as type and enter an arbitrary name.
- The new RecordContainer will appear as sub-node of node "RecordContainers". Click on it and edit its properties:
  - recordFieldMappings: Add the EntityFields to be filled by the JDitoRecordContainer. IMPORTANT:
    - The order of the fields will be the order of the data in the array built in the contentProcess (see below).

    - An EntityField named 'UID' (spelled exactly like this!), with contentType TEXT must always be present and included in the record field mapping.

  - contentProcess: This code must return a nested array acting as data source. The order of the data in the array must be exactly the order of the fields in the recordFieldMapping (see above). In principle, the contentProcess looks like this:

    XXX_entity.myJDitoRecordContainer.contentProcess

    ```
    var entityField1value1 = (...);
    var entityField2value1 = (...);
    var entityField3value1 = (...);
    (...)

    var myDataArray = [];
    myDataArray.push([entityField1value1, entityField2value1, entityField3value1]);
    myDataArray.push([entityField1value2, entityField2value2, entityField3value2]);
    myDataArray.push([entityField1value3, entityField2value3, entityField3value3]);

    result.object(myDataArray);
    ```

Example:

Open Turnover_entity (Context "Turnover") in the Navigator window. Click on RecordContainers > jdito: The sub-nodes appear in exactly the order defined in Entity recordFieldMappings (and thus, not in alphabetical order).

Now, look at the contentProcess: The resulting array is included in the variable `chartData`, which is filled in a loop including the following line:

```
// EntityFields: UID, PARENT, CATEGORY, X, Y
chartData.push([key, countDataSet.parent, countDataSet.category, countDataSet.x, countDataSet.count]);
```

Finally, the array is returned: `result.object(chartData);`

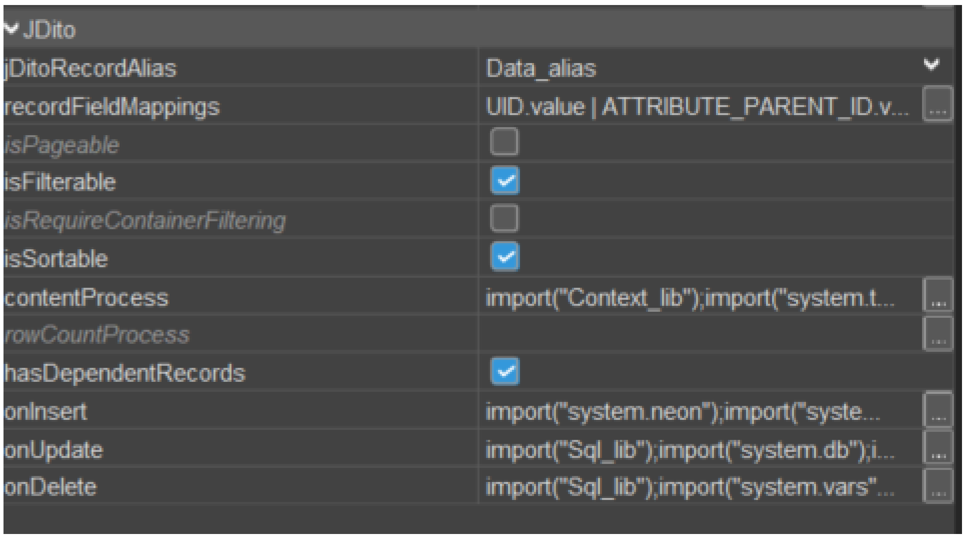
The data retrieved and structured in the Turnover_entity is displayed in the client in various charts, organized in a GroupLayout: Click, e.g., on Sales > Opportunity > MainView > tab Forecast, which includes TurnoverDynamicMultiDataChart_view in the upper right part: Just view the different charts (using the View selection button in the upper right corner).

10.10.2.2. Advanced explanations

The JDitoRecordContainer is one of the currently four types of RecordContainers that serve as a data source of an Entity. Its speciality lies in its flexibility, as the data source is a JDito process (property "contentProcess"). The advantage of increased flexibility comes with the drawback of having to code sorting, paging, and filtering by yourself, within the contentProcess. The source of your data depends on its purpose. You can use the `SqlBuilder` to interface with the database, or you could also use the `net` module to access web services and use those as the source of your data.

If you use a JDitoRecordContainer, be always aware that you need to handle sorting, paging and filtering by yourself. Otherwise it will simply be not supported by your Entity, even if you have activated the corresponding properties.

Important properties



JDito	
jDitoRecordAlias	Data_alias
recordFieldMappings	UID.value | ATTRIBUTE_PARENT_ID.v...
isPageable	☐
isFilterable	✓
isRequireContainerFiltering	☐
isSortable	✓
contentProcess	import("Context_lib");import("system.t...
rowCountProcess	
hasDependentRecords	✓
onInsert	import("system.neon");import("syste...
onUpdate	import("Sql_lib");import("system.db");i...
onDelete	import("Sql_lib");import("system.vars"...

- **jDitoRecordAlias**

  This defines the default alias, which is used by the database access methods in all of the RecordContainer's processes. In many cases, this property will be set to "Data_alias".

- **recordFieldMappings**

  Here you map your EntityFields to the result value of your contentProcess.

  The order of the mapping has to be the same as the order of the arrays returned by your contentProcess.

- **isPageable**

  This determines if paging is active. In your contentProcess you will then get access to `$local.page` (page to be loaded) and `$local.pagesize` (number of datasets per page to be returned).

- **isFilterable**

This property determines if your Entity is filterable or not. If active, you get access to `$local.filter`, which consists of a map that contains the field, the operator, and the value.

- **isRequireContainerFiltering**

  This informs your RecordContainer that filtering should be done serverside. Without this, the result is filtered by the receiving client. If you deal with a large number of datasets, this can give a big boost to performance.

- **isSortable**

  This turns sorting on. You get access to `$local.order`, which contains a map consisting of the fieldname as key and the sorting direction as value.

- **contentProcess**

  This is the actual data source of your RecordContainer. In this process, you have to gather your data and at the end return it as a twofold nested array. For example:

```
var data = [
    ["UID1","VALUE1.1","VALUE2.1","VALUE3.1"]
    ,["UID2","VALUE1.2","VALUE2.2","VALUE3.2"]
    ,["UID3","VALUE1.3","VALUE2.3","VALUE3.3"]
];

result.object(data);
```

- **rowCountProcess**

  This process is used to determine the number of datasets. If it is missing, the contentProcess is executed twice, which can lead to a performance loss, if the contentProcess involves extensive data manipulation in order to generate the data. If you can determine the number of datasets in an easier way, you should do so here.

- **hasDependentRecords**
  If your datasets are interdependent, e.g., in a parent-child structure for trees, then you should check this flag. In particular, this flag has effects when deleting datasets: If it is set to true, then the contentProces will always run after every deletion and thus update (rebuild, refresh) all data and their structure correctly.

- **onInsert**

  This process is used, if you add new datasets to the Entity. Here you should handle how your data is saved. The process is executed per data row. You get access to `$local.rowdata`, which contains the data of the data row. For example:

```
var rowdata = vars.get("$local.rowdata");
var columns = [
    "UID"
    ,"C1"
    ,"C2"
    ,"C3"
];
var values = [
    rowdata["UID.value"]
    , rowdata["C1.value"]
    , rowdata["C2.value"]
    , rowdata["C3.value"]
];

new SqlBuilder().insertData("YOURTABLE", columns, null,
values);
```

> In the onInsert process, do not access EntityField values via `$field` variables, as these may contain outdated values at that time. Use `$local.rowdata` or `$local.initialRowdata` instead (see chapter $local.rowdata and $local.initialRowdata and appendix $local variables).

- **onUpdate**

This process handles the edit of data. Here you have access to `$local.changed`, which contains an array holding all changed EntityFields. For getting the data, `$local.rowdata` is provided, too. To get the data for the update, you have to loop over the array you got from `$local.changed` and use these as index to access `$local.rowdata`. The UID of the row to be updated can be accessed by reading from `$local.uid`

```
var changedFields = vars.get("$local.changed");
var rowData = vars.get("$local.rowdata");

var columns = [];
var data = [];

for(let field in changedFields)
{
    // According to the spelling guidelines (see AID001),
    // EntityFields that represent database columns
    // should be named like the database columns.
    // This enables us to just split the field
    // identifier "NAME.value" at the dot to get the name
    // of the database column at index 0.
    columns.push(changedFields[field].split(".")[0] );
    data.push( rowData[changedFields[field]] );
}

newWhereIfSet("YOURTABLEID = '" + vars.get("$local.uid") + "'")
.updateData(true, "YOURTABLE", columns, null, data);
```

In the onUpdate process, do not access EntityField values via `$field` variables, as these may contain outdated values at that time. Use `$local.rowdata` or `$local.initialRowdata` instead (see chapter $local.rowdata and $local.initialRowdata and appendix $local variables).

- **onDelete**

This is the process that handles the deletion of data. Here you only get the variable `$local.uid` to identify the dataset that is to be deleted.

```
newWhereIfSet("YOURTABLEID = '" + vars.get("$local.uid") + "'")
.deleteData(true, "YOURTABLE");
```

In the onDelete process, do not access EntityField values via `$field` variables, as these may contain outdated values at that time.

The code examples above are assuming you are using a database as your data source.
If you want to use a web service, you have to design the properties onInsert, onUpdate, and onDelete accordingly, in order to send the new/changed data back to the web service.

10.10.2.3. Step-by-step example

Now, for learning and testing purposes, let’s build our own Entity with a JDitoRecordContainer step-by-step.

At first, we create a new database table with some columns:

Table name: MYTEST

Columns:

- MYTESTID: char(36)
- MYNUMBERFIELD: int
- MYTEXTFIELD: varchar(50)

As usual, we update the Alias Definition in order to have the new table available in ADITO.

Then, we create a new Context "MyTest", an Entity and EntityFields according to the naming conventions - with one exception: The EntityField holding the primary key is only named "UID".

MyTest_entity

- UID
- MYNUMBERFIELD: contentType = number
- MYTEXTFIELD

Create a FilterView, a PreviewView and an EditView, and set the EntityFields and properties accordingly. (We do not need a MainView for our example.)

Add the new Context to a suitable place in the Global Menu (application > _SYSTEM_APPLICATION_NEON > …).

Now we are ready to create the JDitoRecordContainer: Open the Entity in the Navigator window and right-click on the Entity’s name. Choose "New RecordContainer". In the following dialog, select type "jDitoRecordContainer" and enter simply "jDito" as name. Then, a new folder "RecordContainers" will appear, with sub-node "{} jDito" in it.

Click on "jDito", in order to set its properties. For this test example, we will only set some of the properties.

- jDitoRecordAlias: Data_alias
- recordFieldMappings (to simplify matters, we skip the display values):
    - UID.value
    - MYNUMBERFIELD.value
    - MYTEXTFIELD.value

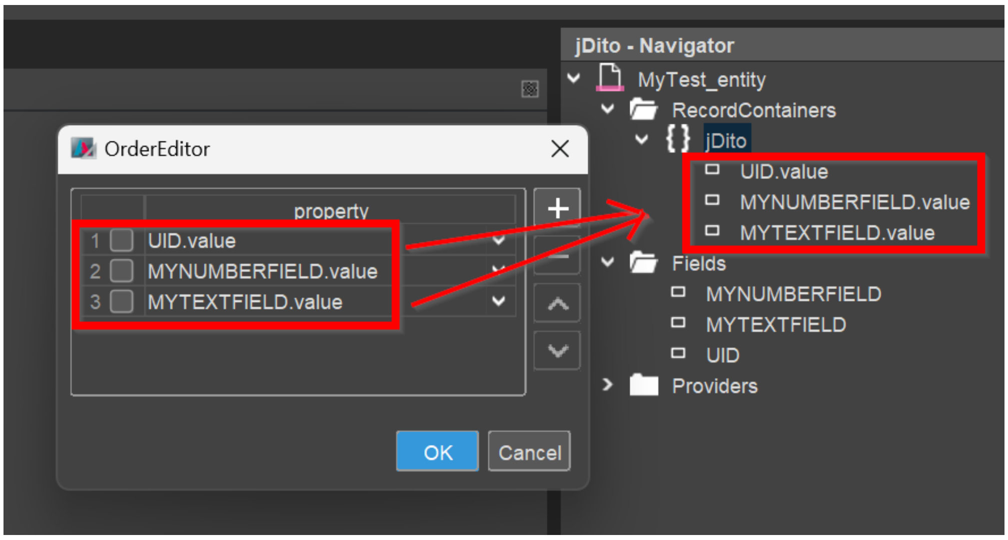


Figure 28. RecordFieldMappings of a simple JDitoRecordContainer

Always keep in mind the order of the RecordFieldMappings, as this order will be important in most of the processes.

- isfilterable: true
- contentProcess: This is the central process of a JDitoRecordContainer. Here, the data is loaded and (if required) filtered. The variable $local.idvalues contains the id(s) of (if so) selected dataset(s) (row(s)), to be shown in the Preview. The variable $local.filters contains the filter, e.g., set by the client user via the filter component of the FilterView.

```
import { result, vars } from "@aditosoftware/jdito-types";
import { FilterSqlTranslator } from "JditoFilter_lib";
import { newSelect, SqlBuilder } from "SqlBuilder_lib";

var query = newSelect("MYTESTID, MYNUMBERFIELD, MYTEXTFIELD")
    .from("MYTEST");

if (vars.exists("$local.idvalues") && vars.get("$local.idvalues"))
{
    // selected row(s), to be shown in the Preview
    query.whereIfSet("MYTEST.MYTESTID", vars.get("$local.idvalues"), SqlBuilder.IN());
}
 else if (vars.get("$local.filters"))
{
    // load with filter
    var filterCondition = new FilterSqlTranslator(vars.get("$local.filters"), "MYTEST");

    query.whereIfSet(filterCondition.getSqlCondition());
}

var data = query.table();

result.object(data);
```

- onInsert: This is the process to control how new datasets are inserted (saved). Use the EditView for entering example data for MYNUMBERFIELD and MYTEXTFIELD (not for MYTESTID/UID, as this column/field will be automatically handled in the background), and fill the onInsert process accordingly, e.g., like this:

```
import { vars } from "@aditosoftware/jdito-types";
import { SqlBuilder } from "SqlBuilder_lib";

var rowdata = vars.get("$local.rowdata");

// The columns' order must match the order of the values.
var columns = [
    // In the database, the ID column is named
    // according to the naming conventions (see AID001)
    "MYTESTID"
    , "MYNUMBERFIELD"
    , "MYTEXTFIELD"
];

// The values' order must match the order of the columns.
var values = [
    // In an Entity with a JDitoRecordContainer,
    // the ID field must always be named "UID".
    rowdata["UID.value"]
    , rowdata["MYNUMBERFIELD.value"]
    , rowdata["MYTEXTFIELD.value"]
];
new SqlBuilder().insertData("MYTEST", columns, null,
    values);
```

- onUpdate: This process handles the update (change) of existing datasets, e.g., via the EditView. In this example, its code can be kept short:

```
import { vars } from "@aditosoftware/jdito-types";
import { newWhereIfSet } from "SqlBuilder_lib";

var changedFields = vars.get("$local.changed");
var rowData = vars.get("$local.rowdata");

var columns = [];
var data = [];

for(let field in changedFields)
{
    // According to the spelling guidelines (see AID001),
    // EntityFields that represent database columns
    // should be named like the database columns.
    // This enables us to just split the field
    // identifier "NAME.value" at the dot to get the name
    // of the database column at index 0.
    columns.push(changedFields[field].split(".")[0] );
    data.push( rowData[changedFields[field]] );
}

newWhereIfSet("MYTESTID = '" + vars.get("$local.uid") + "'")
.updateData(true, "MYTEST", columns, null, data);
```

- onDelete: This process handles the deletion of datasets. If more than one dataset has been marked, the onDelete process is executed separately for every marked dataset, with variable $local.uid filled accordingly. In many cases, the code of the onDelete process can be kept quite simple, e.g., like this:

```
import { vars } from "@aditosoftware/jdito-types";
import { newWhereIfSet } from "SqlBuilder_lib";

newWhereIfSet("MYTESTID = '" + vars.get("$local.uid") + "'")
.deleteData(true, "MYTEST");
```

This simple step-by-step example should help you to get a little bit more familiar with JDitoRecordContainers, which are, in practice, often way more complex. Indeed, the above example would never be used in practice, as a DbRecordContainer would fit the task better, because it already includes automatisms for, e.g., interpreting the filter.

The power of a JDitoRecordContainer lies in its flexibility, to load or calculate, filter, sort, insert, update and delete data nearly without any restrictions. Otherwise, its disadvantage is its complexity and that you have to care manually for things that are automated in a DbRecordContainer.

A common use case that often requires a JDitoRecordContainer, is a Tree structure.

10.10.2.4. Filtering a JDitoRecordContainer

In the previous chapter you can see already a plain example of integrating a filter in a JDitoRecordContainer. However, to enable also complex filtering, you can find further functions in library JDitoFilter_lib, e.g., the very useful FilterSqlTranslator. These functions are either for filtering the data manually or for building an SQL condition.
In the JDitoRecordContainer of, e.g., the Contexts Attribute, Manager, or Workflow you can find examples of how the various helper functions for building filters are applied.

10.10.3. IndexRecordContainer

In ADITO, the "index" is a kind of parallel data container that can be filled with selected data of connected ADITO databases (e.g., name and address of contact persons or companies). By this data reduction, by a special data structure, and by a special database (comparable with a NoSQL database) the data included in the index can be scanned and read very quickly, using the Apache Solr search engine.

An IndexRecordContainer can be used in various ways, of which the "Global Search" is the most common: If you click on the search button in the web client’s Global Bar, you open a search field, in which you can enter search terms, e.g., the name "Smith"; then, in the result, amongst others, all persons or companies are listed that have a name including "Smith". By clicking on one of them, the corresponding dataset is opened, ready for further processing.

Besides the "Global Search", an IndexRecordContainer can also be used as an alternative data source for EntityFields, if the usage of a DB or JDito RecordContainer is not suitable or does not show the required performance. In the ADITO xRM project, several FilterViews are filled by using an IndexRecordContainer.

To keep the index up-to-date, it is connected to ADITO’s audit process. This process is called for every change in the database. In the "Projects" window, you can find the audit process under process > internal > process_audit.

Please find detailed explanations on the purpose and usage of an IndexRecordContainer in the ADITO Information Document AID093 "Indexsearch".

10.10.4. DatalessRecordContainer

A DatalessRecordContainer makes it possible to use an Entity without loading or writing data via the RecordContainer. This enables the mere entry of data, which can then be read and processed, e.g., via an Action.

Example: If you want to request specific information before opening the actual Context, you could realize this via an Entity having a DatalessRecordContainer. Via an Action, the information could subsequently be extracted from the EntityFields and then be passed to another Entity via method `neon.openContextWithRecipe`, using a Parameter.

Examples in ADITO xRM:

There are several examples of the usage of a DatalessRecordContainer in the ADITO xRM project - just do a full-text search for the term "datalessRecordContainer".

An easy-to-understand example is "BulkMailTesting_entity". This Entity's View "BulkMailTesting_view" is opened, when, in the FilterView of Context "BulkMail", a dataset is selected and Action "Test email" is called via the three-dotted button in the CardViewTemplate of the PreviewView - see BulkMail_entity.testMail.onActionProcess. BulkMailTesting_view is only required for selecting a contact and entering the recipient email address - both are temporary data that does not need to be stored anywhere: After the user has pressed the button "Test email", the data is extracted in process BulkMailTesting_entity.testMail.onActionProcess (do not mistake this process with the onActionProcess quoted before) and used for sending out the test email. If the user had set the switch "Save settings" to true, then CONTACT_ID and email recepient are stored in the database table BULKMAIL - i.e., the table referring to BulkMail_entity. Thus, BulkMailTesting_entity itself does not need database access and therefore uses a DatalessRecordContainer.

You may wonder why a DatalessRecordContainer also features the property "alias" (for specifying the default alias), when in fact there is no database access in this case. Well, this is simply a matter of consistency: "alias" must be a common property of *all* types of RecordContainers. This has technical reasons, because the existence of a RecordContainer with an "alias" property is a prerequisite for all methods that allow database access on Entity level (e.g., in Actions).

If you open the PreviewView of an Entity connected to a DatalessRecordContainer, and there is an Action involved that does not cause a "jump" to another Entity, then please make sure that the PreviewView is closed via the following code line at the end of the onActionProcess:

```
neon.closeImage(vars.get("$sys.currentimage"), true);
```

Otherwise, the PreviewView will remain open, and the user will not get any

> feedback, when the Action is finished, but he must close the window manually.

10.11. Tags

> To understand this chapter, please first read the chapter on the ViewTemplate type Favorite.

Tags are useful if you want to "attach a label" to specific datasets, in order to mark them for various purposes. They will then appear in Context "Favorite" (see "star" button in the sidebar of the client), grouped according to property "Tag", to keep track of these datasets in an ordered way. (Exception: Datasets exclusively tagged by *Hash*tags do not appear in Context "Favorite".)

Some use cases require tags to be set not by the user, via a ViewTemplate of type "Favorite", but via an Action or via an automatism.

In principle, you could customize the assignment of tags simply by creating the required datasets in the system tables ASYS_RECORDGROUP and ASYS_RECORD (see chapter Favorite). However, the preferable way to do this is to use the methods of a library named "tag", which can be imported via `import { tag } from "@aditosoftware/jdito-types";`

There are different methods for tagging and un-tagging, as well as for getting a tag object - but the principle steps are always the same:

1. You create a purpose-specific config object.
2. You execute the actual method for (un-)tagging itself, with the config object as parameter.

Example:
The execution of the following example code (e.g., to be included in the onActionProcess of a test Action) adds a tag titled "Test Tag" to a dataset of Context "Offer", visible in the client of the current user:

```
var userId = tools.getCurrentUser()["name"];

// OFFERID of demo dataset "1004-1" of Offer_entity
var recordId = "ab61911c-88c5-4d79-9ac2-f41f21154dbe";

// creating the config object
var config = tag.createAddTagConfig();
config.setObjectType("Offer");
config.setRowId(recordId);
config.setTagTitle("Test Tag");
config.setTagType(tag.FAVORITE_GROUP);
config.setUserId(userId);

// adding the tag
```

```
tag.add(config);
```

The methods used in this example code should be self-explanatory, if you have read chapter Favorite before.

Depending on the purpose, the config object must be created with one of the following methods:

- `tag.createAddTagConfig()`
- `tag.createGetTagConfig()`
- `tag.createGetTaggedObjectByDataConfig()`
- `tag.createGetTaggedObjectByIdConfig()`
- `tag.createGetTaggedObjectsConfig()`
- `tag.createGetTagsConfig()`
- `tag.createUntagByContextConfig()`
- `tag.createUntagByDataConfig()`
- `tag.createUntagByIdConfig()`
- `tag.createUntagMultipleByIdConfig()`

Each config object has individual configuring methods and is to be set as parameter of one of the corresponding excution methods:

- `add(pConfig)`
- `untag(pConfig)`
- `getTags(pConfig)`
- `getPublicTags(pConfig)`
- `getTaggedObjects(pConfig)`

Besides, there are further utility methods like

- `getTagAlias()`
- `lookupHashtags(pInputPattern)`
- `suggestHashtags(pTagConfig)`

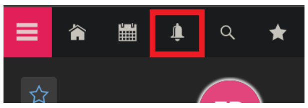
All methods are well-documented via JSDoc, which you can access as usual (via the auto-completion).

10.12. Notifications and observations

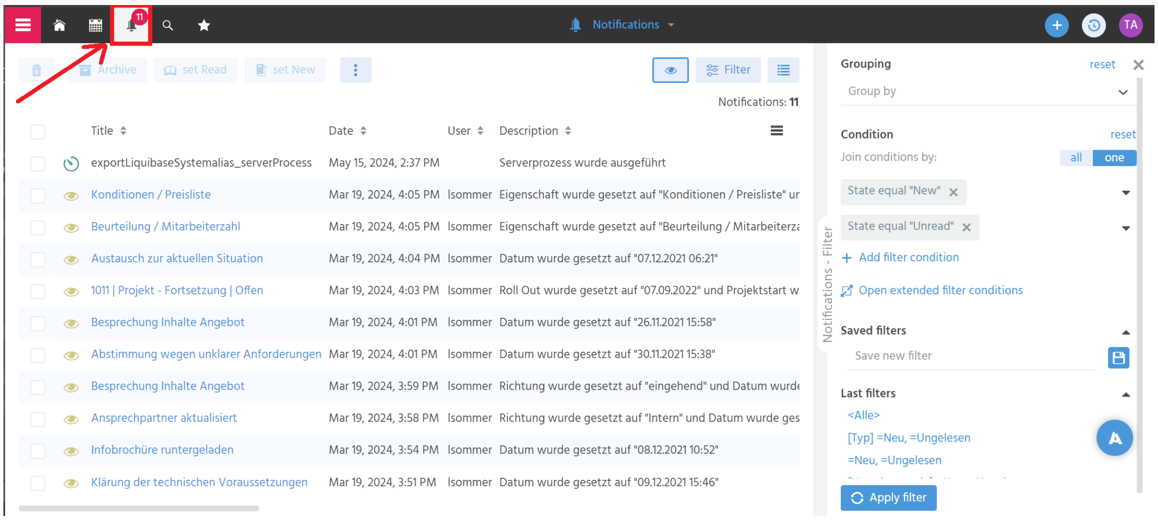
10.12.1. Basics

Notifications are pieces of information shown in ADITO

1. in the NotificationFilter_view, available via the "Bell" icon in the left upper corner of the ADITO web client:

   Here, one or multiple notifications are displayed in a table, with the option to filter them, mark them as "read", etc.:

   Notifications: 11

	Title	Date	User	Description
	exportLiquibaseSystemalias_serverProcess	May 15, 2024, 2:37 PM		Serverprozess wurde ausgeführt
	Konditionen / Preisliste	Mar 19, 2024, 4:05 PM	lsommer	Eigenschaft wurde gesetzt auf "Konditionen / Preisliste" ur
	Beurteilung / Mitarbeiterzahl	Mar 19, 2024, 4:05 PM	lsommer	Eigenschaft wurde gesetzt auf "Beurteilung / Mitarbeiterzа
	Austausch zur aktuellen Situation	Mar 19, 2024, 4:04 PM	lsommer	Datum wurde gesetzt auf "07.12.2021 06:21"
	1011 | Projekt - Fortsetzung | Offen	Mar 19, 2024, 4:03 PM	lsommer	Roll Out wurde gesetzt auf "07.09.2022" und Projektstart w
	Besprechung Inhalte Angebot	Mar 19, 2024, 4:01 PM	lsommer	Datum wurde gesetzt auf "26.11.2021 15:58"
	Abstimmung wegen unklarer Anforderungen	Mar 19, 2024, 4:01 PM	lsommer	Datum wurde gesetzt auf "30.11.2021 15:38"
	Besprechung Inhalte Angebot	Mar 19, 2024, 3:59 PM	lsommer	Richtung wurde gesetzt auf "eingehend" und Datum wurde
	Ansprechpartner aktualisiert	Mar 19, 2024, 3:58 PM	lsommer	Richtung wurde gesetzt auf "Intern" und Datum wurde ges
	Infobrochüre runtergeladen	Mar 19, 2024, 3:54 PM	lsommer	Datum wurde gesetzt auf "08.12.2021 10:52"
	Klärung der technischen Voraussetzungen	Mar 19, 2024, 3:51 PM	lsommer	Datum wurde gesetzt auf "09.12.2021 15:46"

   Grouping — reset
   Group by

   Condition — reset
   Join conditions by: all [✓ one]
   State equal "New"
   State equal "Unread"
   + Add filter condition
   Open extended filter conditions

   Saved filters
   Save new filter

   Last filters
   <Alle>
   [Typ] =Neu, =Ungelesen
   =Neu, =Ungelesen

   Apply filter

2. via a popup appearing in the lower right corner of the web client (once per one single notification), given that you had once accepted being notified by ADITO

Notifications can be triggered in various ways, in particular via

- changes of data that is covered by "observation". Observations can be set by the web client user, in various ways - find more information in the ADITO end user training course "Notifications and observations" and the corresponding documentation.
- storing datasets that include hyper-references to a specific ADITO user (Employee), e.g., "(…) @J.Smith (…)"
- incoming telephone calls - find extensive information in the ADITO Information Document AID018 "CTI".
- manual triggering of a notificaton via JDito

10.12.2. Setup

Every new ADITO cloud system comes with a ready-to-use notification functionality. This includes a notification queue, in which all notifications remain (e.g., as long as someone is offline), until it is possible to deliver them to the respective users (recepients).

Normally, you do not need to customize anything to use notifications. (In case you have an earlier ADITO system without notification functionality, contact ADITO for further instructions.) Also, the triggering of notifications via hyper-references is already built-in and does not need to be configured or activated. The setup of CTI is explained in the ADITO Information Document AID018 "CTI".

Thus, the only functionalities that require customizing are manual notification triggers and observation.

10.12.2.1. Manually triggered notifications

You can arbitrarily trigger a notification manually via JDito, by using the backend methods of class `notification`. There are 2 steps:

1. Create and set a configuration object
2. Execute the notification trigger, using the configuration object - with "execute" actually meaning to add a notification to the list (queue) of notifications to be sent to the user(s)

Here is an example, included in the ADITO xRM project. You may use this example as a pattern to design your own notification triggers:

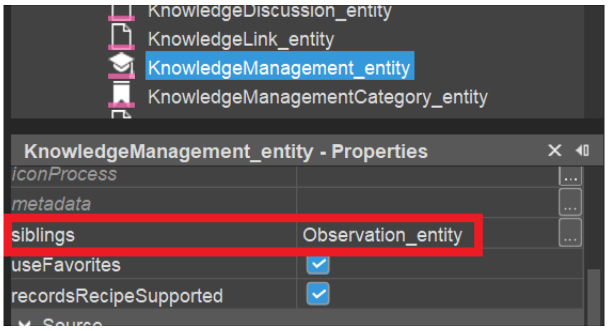
KnowledgeManagement_entity.Likes.onActionProcess.js

```
(...)
let notificationConfig = notification.createConfig()
    .addUserWithId(employeeUserId)
    .forcedPriority(notification.PRIO_LOW)
    .notificationType("Like")
    .initialState(notification.STATE_UNSEEN)
    .caption(caption)
    .description(description);

notification.addNotificationWith(notificationConfig);
(...)
```

10.12.2.2. Observation

If you want to augment an Entity with the option to "observe" one or multiple datasets of it, you need to add specific Actions and configure them accordingly.

As a first step, add Observation_entity as sibling to the Entity that is to be observed. See, e.g., KnowledgeManagement_entity:


KnowledgeDiscussion_entity
KnowledgeLink_entity
KnowledgeManagement_entity
KnowledgeManagementCategory_entity
KnowledgeManagement_entity - Properties

iconProcess	
metadata	
siblings	Observation_entity
useFavorites	✓
recordsRecipeSupported	✓

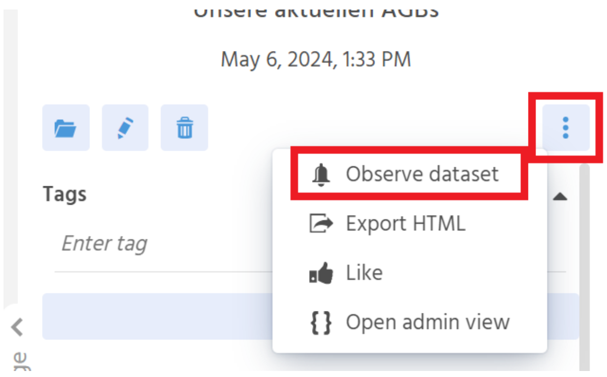
Setting this sibling makes sure that the corresponding actions (see below) will be updated automatically. (Otherwise, e.g., it would be possible that an observation can be added multiple times.)

Furthermore, in the RecordContainer’s process onDBInsert/OnDBUpdate/onDBDelete you need to add the function `EventHandler.onInsert()/EventHandler.onUpdate()/EventHandler.onDelete()`.

Then, open Dependency_lib (in the project tree, under process > libraries) and add the connection of the dependent Entities.

10.12.2.2.1. Observation of selected datasets

Selected datasets can be observed (and "un-observed") via a corresponding Action in the PreviewView, executed by the web client user:



To include this functionality, proceed as follows:

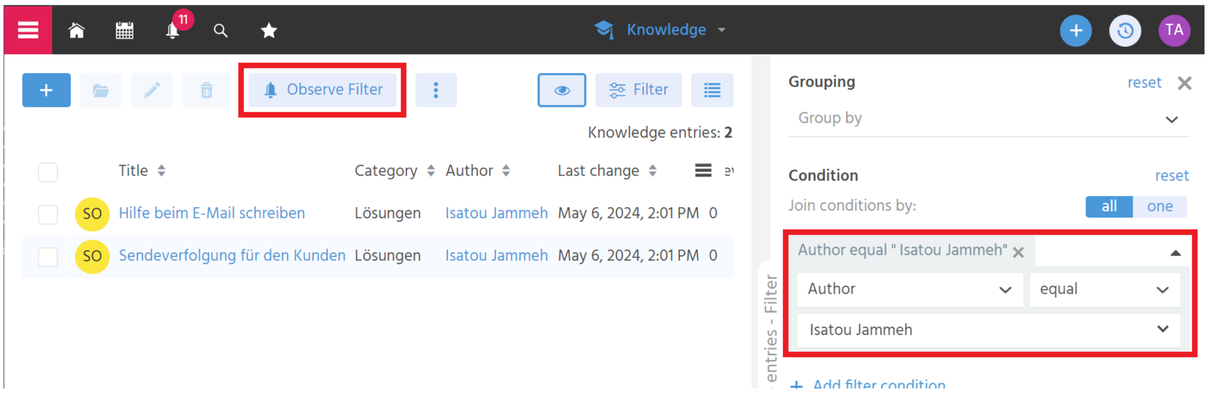
1. Add an ActionGroup named "observeActionGroup".
2. Under this ActionGroup, add 2 Actions named "observe" and "cancelObservation".
3. Configure these Actions' properties, following the pattern given in several Entities of the ADITO

xRM project, e.g., KnowledgeManagement_entity.

> An even faster approach is to simply
>
> - copy & paste this ActionGroup from another Entity of the ADITO xRM project, e.g., KnowledgeManagement_entity;
> - adapt parameter "pUid" of method `Observation.actionStateRecordsRecipe` in property "stateProcess"

10.12.2.2.2. Observation of filtered datasets

Selected datasets can be observed (and "un-observed") via a corresponding Action in the FilterView, executed by the web client user:



	Title	Category	Author	Last change	
SO	Hilfe beim E-Mail schreiben	Lösungen	Isatou Jammeh	May 6, 2024, 2:01 PM	0
SO	Sendeverfolgung für den Kunden	Lösungen	Isatou Jammeh	May 6, 2024, 2:01 PM	0

To include this functionality, proceed as follows:

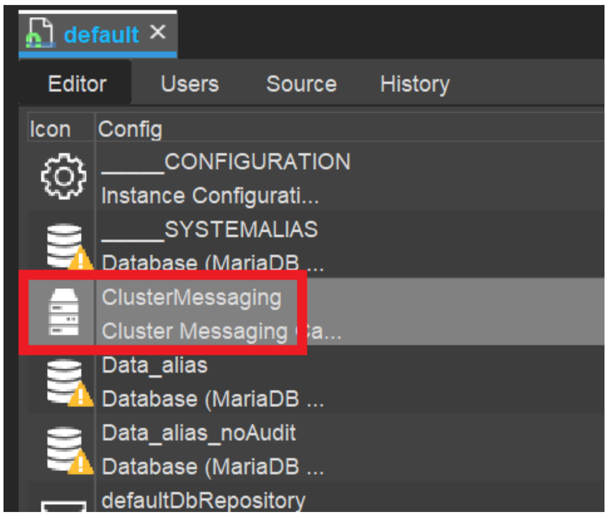
- Add the functionality for observing selected datasets, as explained in chapter Observation of selected datasets
- Select ActionGroup "observeActionGroup" as favoriteActionGroup of the TableViewTemplate (or TreeTableViewTemplate etc.) of your FilterView.

10.12.3. Notifications with multiple ADITO servers

If your system includes multiple ADITO servers, it would be negative, if each server managed only his own notifications, independently from the notifications triggered on the other servers. In this case, server A had no information what happens on server B, and vice versa.

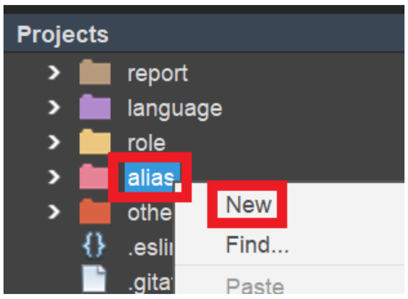
Example:
Each managed ADITO cloud system consists of at least one background server and one foreground server. Now, if a CTI telephone call came in at the background server, the respective user would get no notification, as they is logged into the foreground server.

To avoid this, a so-called cluster messaging server needs to be applied, in order to ensure a distributed notification management. Therefore, every managed ADITO cloud system comes with an installation of Apache Ignite as pre-configured, ready-to-use cluster messaging server. (Besides, ADITO utilizes Ignite also as remote cache server, see chapter Shared caching with multiple ADITO servers.) Its alias has the type "Cluster Messaging", see AliasConfig:

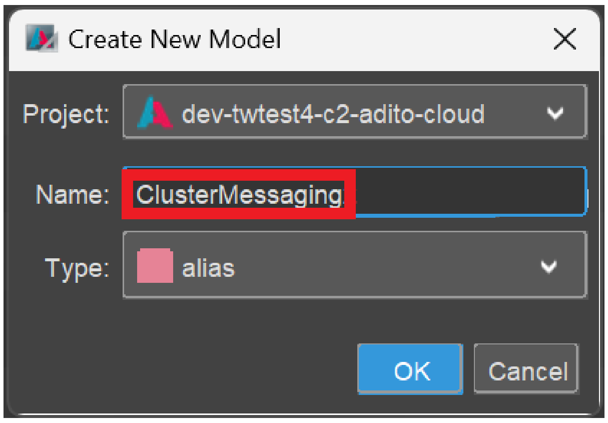


If this alias is not present yet, you need to add it first:

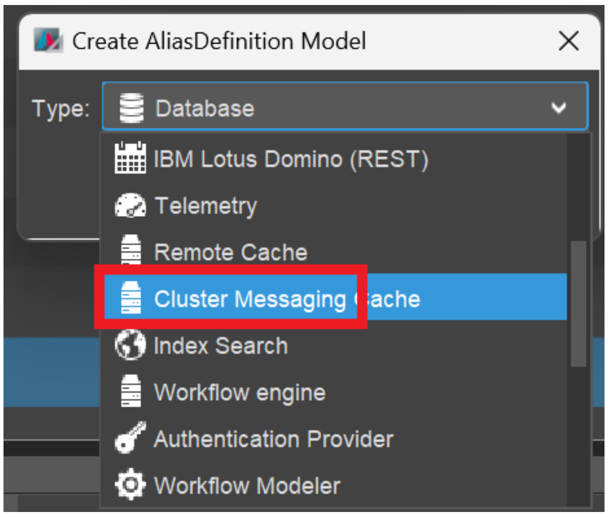
1. In the project tree, right-click on node "alias" and choose "New" from the context menu.



2. A dialog named "Create New Model" appears. Here, type in a suitable name (e.g., "ClusterMessaging").

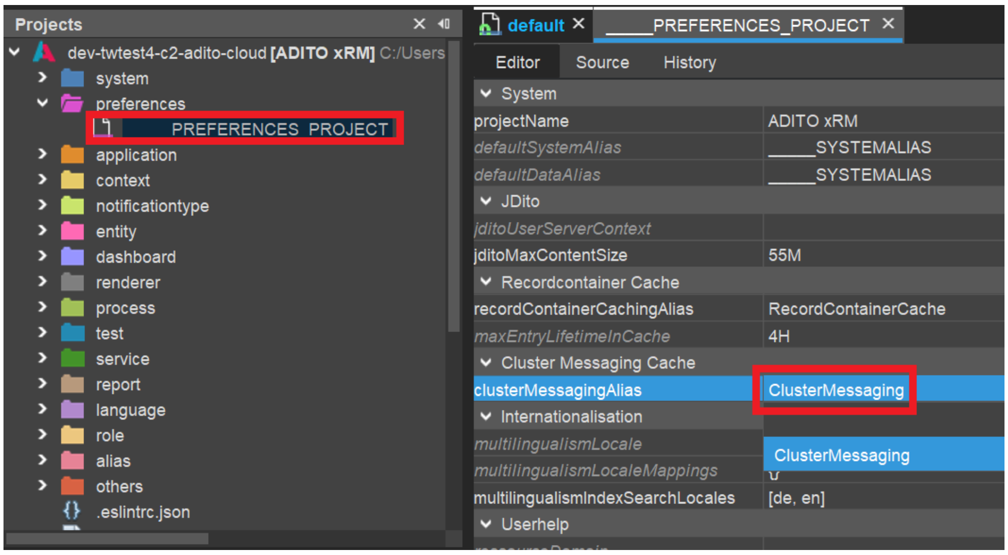


3. A dialog named "Create AliasDefinition Model" appears. Here, select the type "Cluster Messaging".



4. Deploy your project. Then, the new alias appears in the AliasConfig.

Now, check if the cluster messaging alias is set as value of the project property "clusterMessagingAlias" (see preferences > ____PREFERENCES_PROJECT, in the project tree):

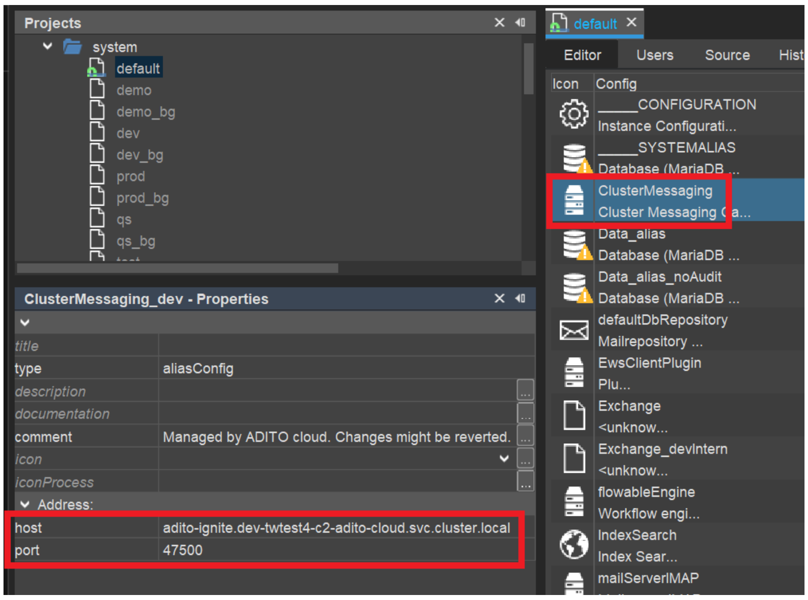


System	
projectName	ADITO xRM
defaultSystemAlias	____SYSTEMALIAS
defaultDataAlias	____SYSTEMALIAS
JDito	
jditoUserServerContext	
jditoMaxContentSize	55M
Recordcontainer Cache	
recordContainerCachingAlias	RecordContainerCache
maxEntryLifetimeInCache	4H
Cluster Messaging Cache	
clusterMessagingAlias	ClusterMessaging
Internationalisation	
multilingualismLocale	ClusterMessaging
multilingualismLocaleMappings	
multilingualismIndexSearchLocales	[de, en]
Userhelp	

If it is not set yet,

1. set it now,
2. deploy your project,
3. restart the ADITO server,
4. re-establish the tunnel to your cloud system,
5. reconnect to your your system, in order to see the AliasConfig again.

Now, if you click on the cluster messaging alias in the AliasConfig, you can inspect its properties in the "Properties" window. Here, you should see that the address of the cluster messaging server (properties "host" and "port") has been set automatically:



ClusterMessaging_dev - Properties

title	
type	aliasConfig
description	
documentation	
comment	Managed by ADITO cloud. Changes might be reverted.
icon	
iconProcess	
Address:	
host	adito-ignite.dev-twtest4-c2-adito-cloud.svc.cluster.local
port	47500

____CONFIGURATION	Instance Configurati...
____SYSTEMALIAS	Database (MariaDB ...
ClusterMessaging	Cluster Messaging Ca...
Data_alias	Database (MariaDB ...
Data_alias_noAudit	Database (MariaDB ...
defaultDbRepository	Mailrepository ...
EwsClientPlugin	Plu...
Exchange	<unknow...
Exchange_devIntern	<unknow...
flowableEngine	Workflow engi...
IndexSearch	Index Sear...
mailServerIMAP	

This semi-automatic activation of a cluster messaging server only works for managed ADITO cloud systems, as these systems by default come with a pre-configured, ready-to-use installation of a cluster messaging server. If, however, your system is an unmanaged cloud system, you first need to order the transformation of your system to a managed cloud system from ADITO.

ADITO does not offer support of integrating cluster messaging servers into "on premise" (not cloud-based) systems. Although, in principle, this is possible, the installation and integration of a cluster messaging server must be realized by the customers themselves.

10.13. Adding an ATTRIBUTES tab

Several of the ADITO xRM-Project’s Contexts have a tab named "ATTRIBUTES" available in their MainView. Attributes are specific features that can be assigned to certain datasets.

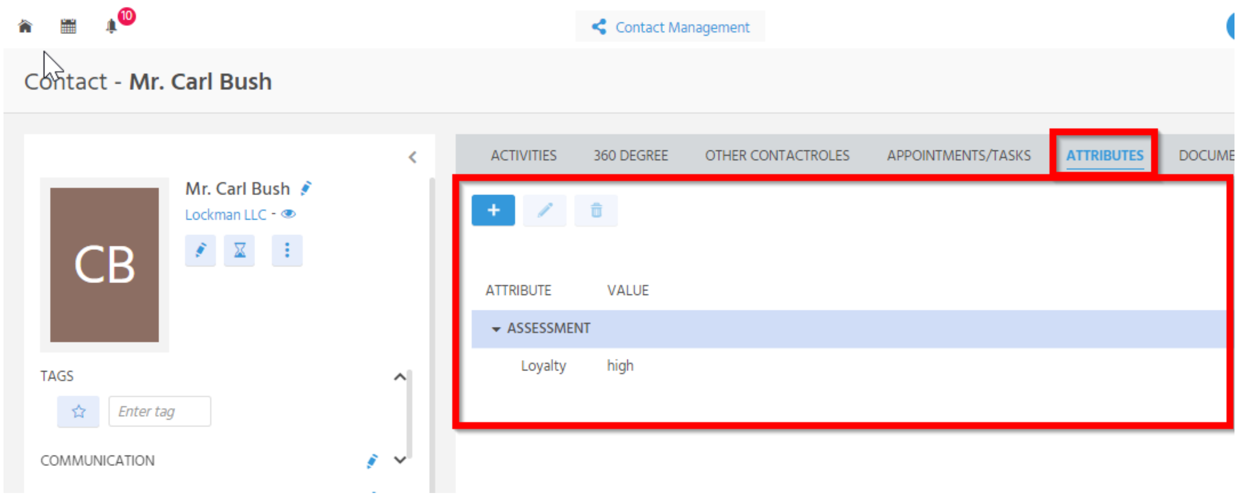


Figure 29. Example of an "ATTRIBUTES" tab, in PersonMain_view

These are the steps to add a similar ATTRIBUTES tab to the MainView of another Context:

1. Create 2 Consumers:
   a. one Consumer that should be named "Attributes" and includes the following property settings:
      - entityName: AttributeRelation_entity
      - fieldName: AttributeRelations
      - onValidation: cp. Person_entity:

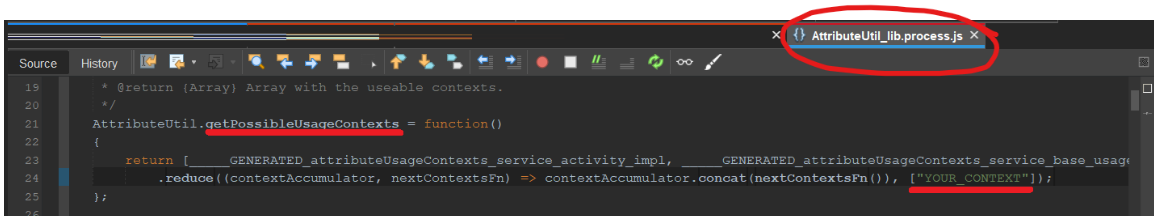
        ```
        result.string(AttributeRelationUtils.validateAttributeCount(vars.get("$field.CONTACTID"), ContextUtils
        .getCurrentContextId(), "Attributes"));
        ```

        (replace `CONTACTID` by the EntityField that is related to the Context’s primary key)
   b. one Consumer that should be named "AttributeTree" and includes the following property settings:
      - entityName: AttributeRelation_entity
      - fieldName: TreeProvider
   c. For both Consumers, fill Parameters ObjectRowId_param and ObjectType_param

2. Adapt Views:
    a. EditView: Assign View reference to AttributeRelationMultiEdit_view (via Consumer "Attributes" , see above)
    b. MainView: Assign View reference to AttributeRelationTree_view (via Consumer "AttributeTree" , see above)
3. Create FilterExtensionSet "Attribute_filter", if the Attributes should be filterable (see chapter FilterExtensionSet and use, e.g., Person_entity as pattern).
4. In the Entity's afterUiInit property, place the code required for the automatic presetting of the Attributes when a new dataset is being created - see, e.g., Person_entity

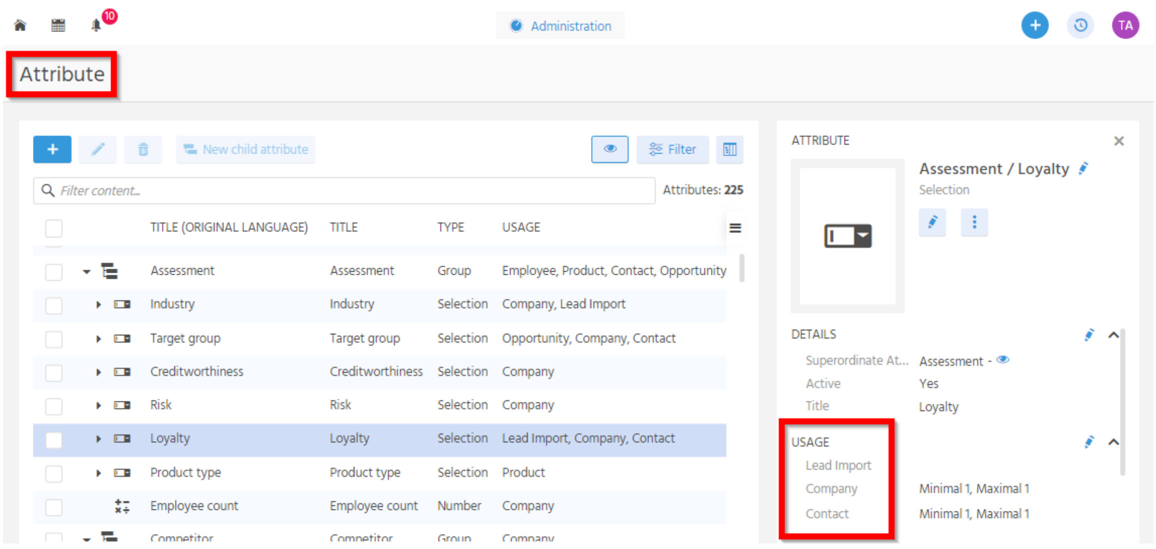
```
if(vars.get("$sys.recordstate") == neon.OPERATINGSTATE_NEW)
{
   AttributeRelationUtils.presetMandatoryAttributes(ContextUtils.getCurrentContextId(), "Attributes");
}
```

5. Add your Context in method `AttributeUtil.getPossibleUsageContexts()` (in library `AttributeUtil_lib`):



```
 * @return {Array} Array with the useable contexts.
 */
AttributeUtil.getPossibleUsageContexts = function()
{
    return [_____GENERATED_attributeUsageContexts_service_activity_impl, _____GENERATED_attributeUsageContexts_service_base_usage
        .reduce((contextAccumulator, nextContextsFn) => contextAccumulator.concat(nextContextsFn()), ["YOUR_CONTEXT"]);
};
```

6. In the Web Client, navigate to Context "Attributes" (in menu group "Administration") and add the Context to section "Usage" of every Attribute that you want to be available in your Context (or create a new Attribute for you Context first, respectively).
   Example:



Attribute

Attributes: 225

TITLE (ORIGINAL LANGUAGE)	TITLE	TYPE	USAGE
Assessment	Assessment	Group	Employee, Product, Contact, Opportunity
Industry	Industry	Selection	Company, Lead Import
Target group	Target group	Selection	Opportunity, Company, Contact
Creditworthiness	Creditworthiness	Selection	Company
Risk	Risk	Selection	Company
Loyalty	Loyalty	Selection	Lead Import, Company, Contact
Product type	Product type	Selection	Product
Employee count	Employee count	Number	Company
Competitor	Competitor	Group	Company

ATTRIBUTE

Assessment / Loyalty
Selection

DETAILS

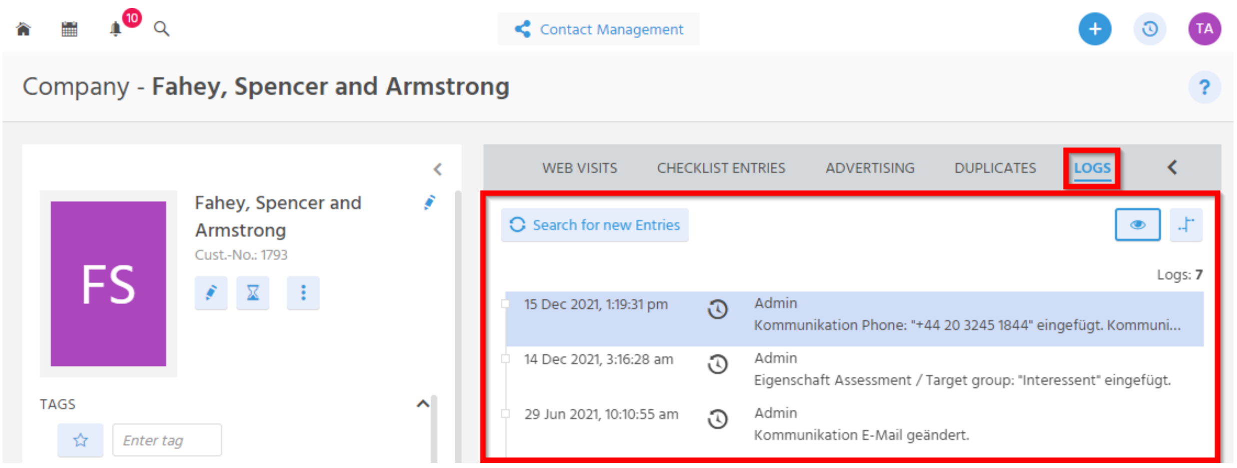
Superordinate At...	Assessment -
Active	Yes
Title	Loyalty

USAGE

Lead Import	
Company	Minimal 1, Maximal 1
Contact	Minimal 1, Maximal 1

10.14. Adding a LOGS tab

Several of the ADITO xRM-Project’s Contexts have a tab named "LOGS" available in their MainView. This kind of logging is not directly related to changes of EntityField values, but to changes of the content of the corresponding database columns (i.e., it is not available for calculated EntityFields that are not mapped to database columns).



Logs: 7

15 Dec 2021, 1:19:31 pm — Admin
Kommunikation Phone: "+44 20 3245 1844" eingefügt. Kommuni...

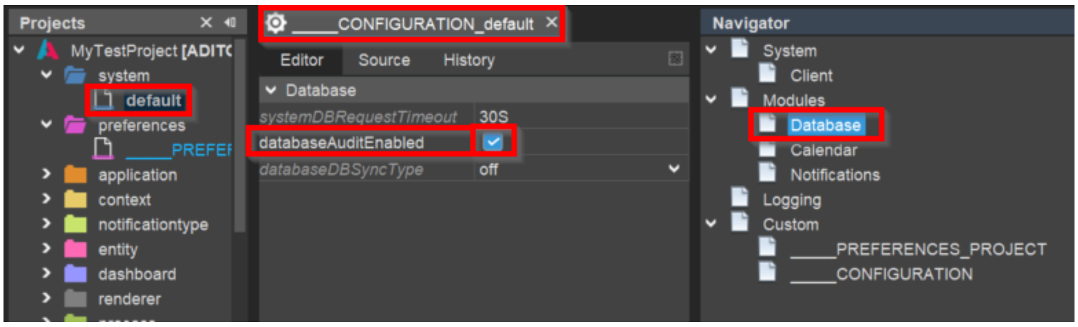
14 Dec 2021, 3:16:28 am — Admin
Eigenschaft Assessment / Target group: "Interessent" eingefügt.

29 Jun 2021, 10:10:55 am — Admin
Kommunikation E-Mail geändert.

Figure 30. Example of a "LOGS" tab, in OrganisationMain_view

These are the steps to add a similar LOGS tab to the MainView of another Context:

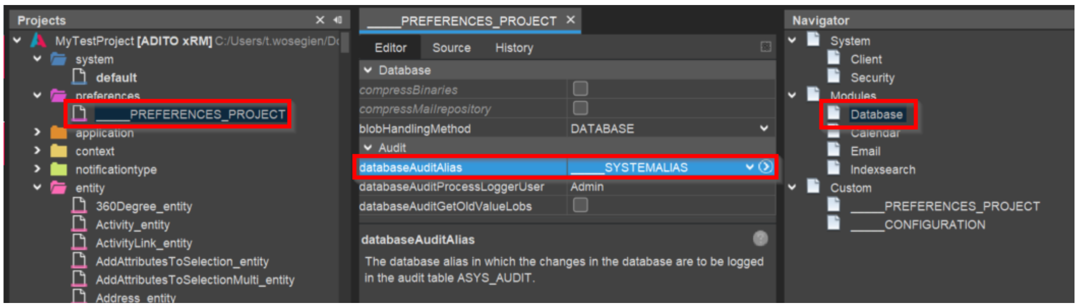
- Prerequisites:
  - Logging must be generally enabled for the ADITO system, via setting property "databaseAuditEnabled" to "true". You can find this property, if (in the "Projects" window) you double-click on system > default and then double-click on "____CONFIGURATION". Then, in the "Navigator" window, navigate to Modules > Database.

    systemDBRequestTimeout 30S
    databaseAuditEnabled ✓
    databaseDBSyncType off

    Restart the server.

  - Logging must be generally enabled for the project, via setting property "databaseAuditAlias" to "__*SYSTEMALIAS". You can find this property, if (in the "Projects" window) you open node "preferences" and double-click on* __PREFERENCES_PROJECT. Then, in the "Navigator" window, navigate to Modules > Database.



Database

compressBinaries	☐
compressMailrepository	☐
blobHandlingMethod	DATABASE

Audit

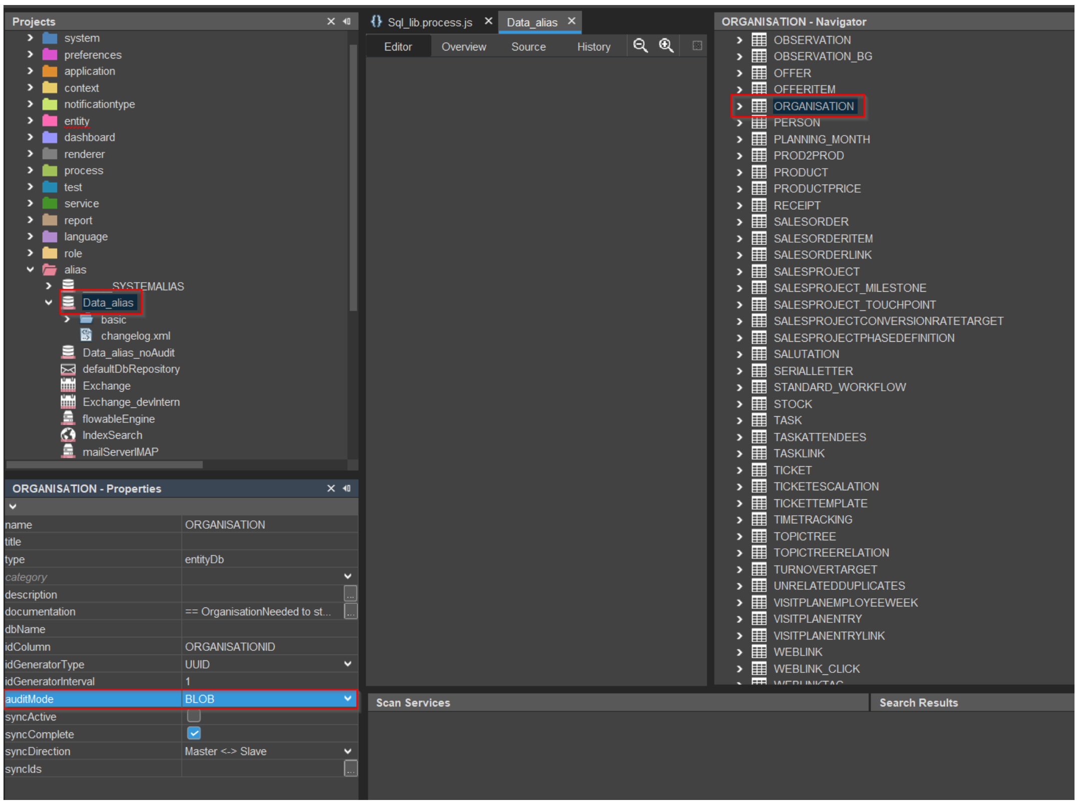
databaseAuditAlias	____SYSTEMALIAS
databaseAuditProcessLoggerUser	Admin
databaseAuditGetOldValueLobs	☐

databaseAuditAlias

The database alias in which the changes in the database are to be logged in the audit table ASYS_AUDIT.

Restart the server.

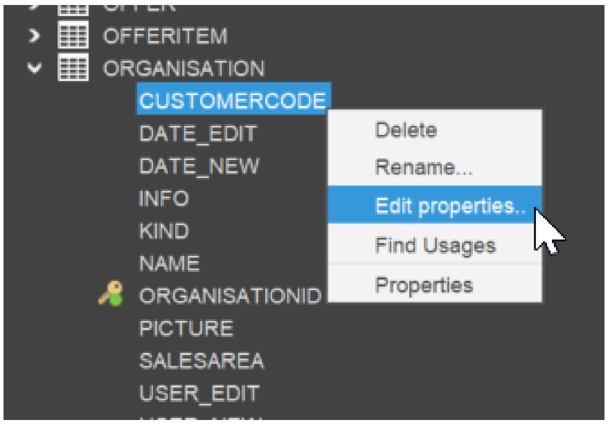
- Logging must be enabled for the respective database table(s), by setting its property "auditMode" to BLOB (in the Alias Definition):



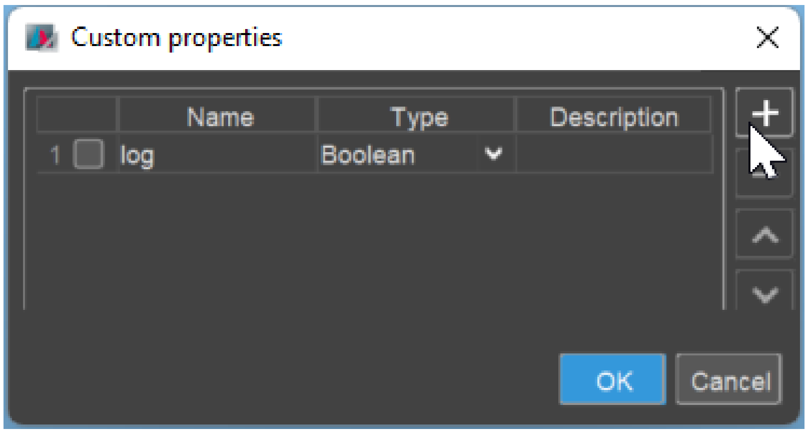
ORGANISATION - Properties

name	ORGANISATION
title	
type	entityDb
category	
description	
documentation	== OrganisationNeeded to st...
dbName	
idColumn	ORGANISATIONID
idGeneratorType	UUID
idGeneratorInterval	1
auditMode	BLOB
syncActive	☐
syncComplete	✓
syncDirection	Master <-> Slave
syncIds	

- Logging must be enabled for every database column that is to be logged, e.g., for table ORGANISATION's column CUSTOMERCODE. This requires the following steps:
  - Right-click on the database column in the Alias Defintion and choose option "Edit properties":

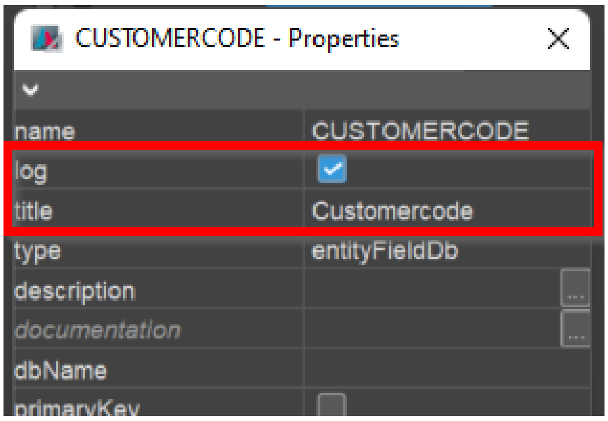


- Add a "custom property" via the "plus" button, name it "log" (exactly spelled like this!) and set its type BOOLEAN:



	Name	Type	Description
1	log	Boolean	

- Then, a new boolean property appears in the property sheet of the column. (If not, simply click on another column and than back again.) Set this property to "true" and set the title to be used for the log entries:



name	CUSTOMERCODE
log	✓
title	Customercode
type	entityFieldDb
description	
documentation	
dbName	
primaryKey	

- Don’t forget to deploy all changes and (as for the project/system-wide settings) restart the server.

    - The corresponding Entity must have the "standard" EntityFields DATE_EDIT, DATE_NEW, USER_EDIT und USER_NEW.

- Create a Consumer (usually named "LogHistories").
- Set the new Consumer’s properties as follows:
    - entityName: LogHistory_entity
    - fieldName: LogHistoryProvider
- Double-click on the new Consumer’s Parameter "tablenames_param" and set the Parameter valueProcess according to this pattern:

```
var res = [];
res.push({id: vars.get("$field.MYIDFIELD"), tableNames: ["MYTABLENAME"]});
res = JSON.stringify(res);
result.object(res);
```

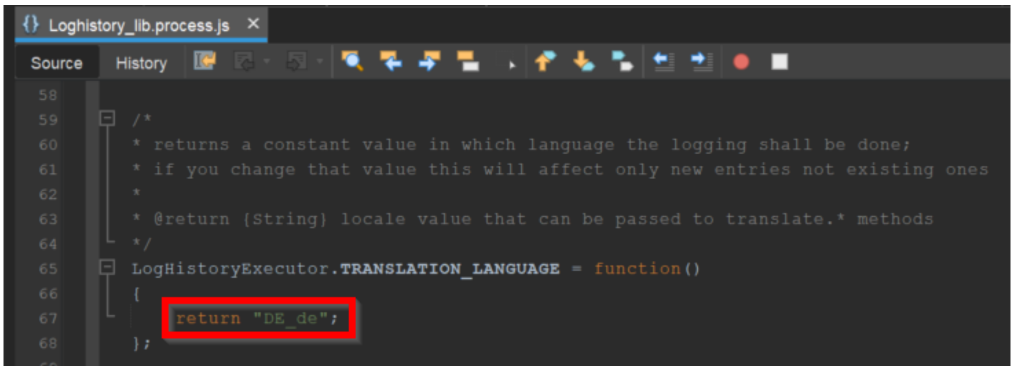
  In this pattern, "MYIDFIELD" and "MYTABLENAME" are, of course, placeholders that must be replaced by the actual name of the related database table(s) and by the name of the EntityField related to the respective database table’s primary key.

  Here is an example of Organisation_entity:

```
var res = [];
res.push({id: vars.get("$field.CONTACTID"), tableNames: ["CONTACT", "COMMUNICATION", "ADDRESS", "AB_ATTRIBUTERELATION",
"COMMUNICATIONSETTINGS"]});
res.push({id: vars.get("$field.ORGANISATIONID"), tableNames: ["ORGANISATION"]});

res = JSON.stringify(res);//currently only strings  can be passed as param
result.object(res);
```

- Properties "expose" and "mandatory" of the new Consumer’s Parameter "tablenames_param" must both remain in default state "true".
- In the "Projects" window, double-click on the MainView that should get the LOGS tab.
- In the "Navigator" window, right-click on the MainView and choose "Add reference to existing View".
- As "EntityField", choose the new Consumer that you had created before (see above).
- As "View", choose LogHistoryFilter_view.
- Click OK, deploy - done!

You can set the language of the log entries in Loghistory_lib (in the project tree, under process > libraries):



```
/*
 * returns a constant value in which language the logging shall be done;
 * if you change that value this will affect only new entries not existing ones
 *
 * @return {String} locale value that can be passed to translate.* methods
 */
LogHistoryExecutor.TRANSLATION_LANGUAGE = function()
{
    return "DE_de";
};
```

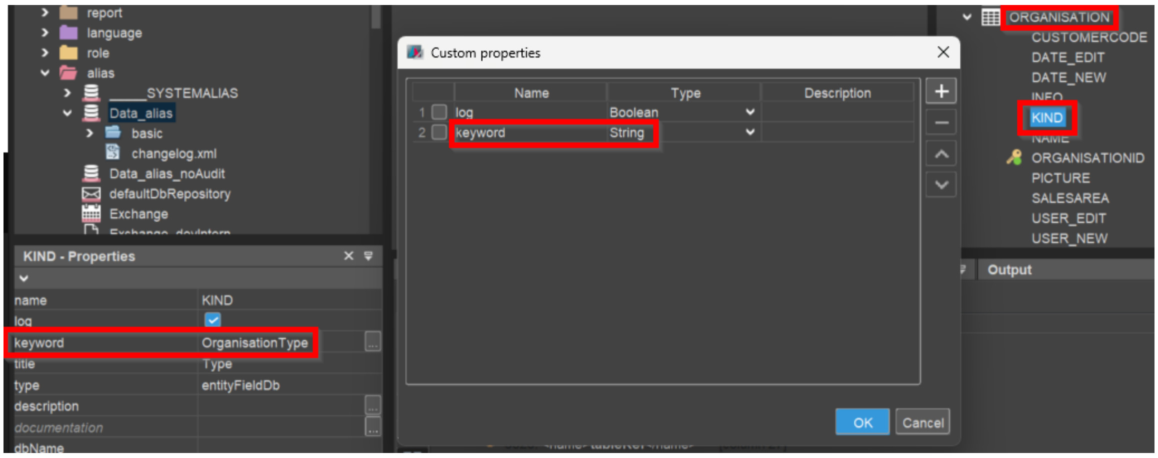
Changing this value will only effect future log entries. Existing log entries will always remain unchanged.

On request, ADITO can provide you with a "Blueprint" that facilitates adding a LOGS tab.

Further useful custom properties:

Besides the custom property "log" (see above) there are further custom properties, which need to be set for specific use cases:

- "keyword" (Type: String); purpose: resolving keywords. Example:



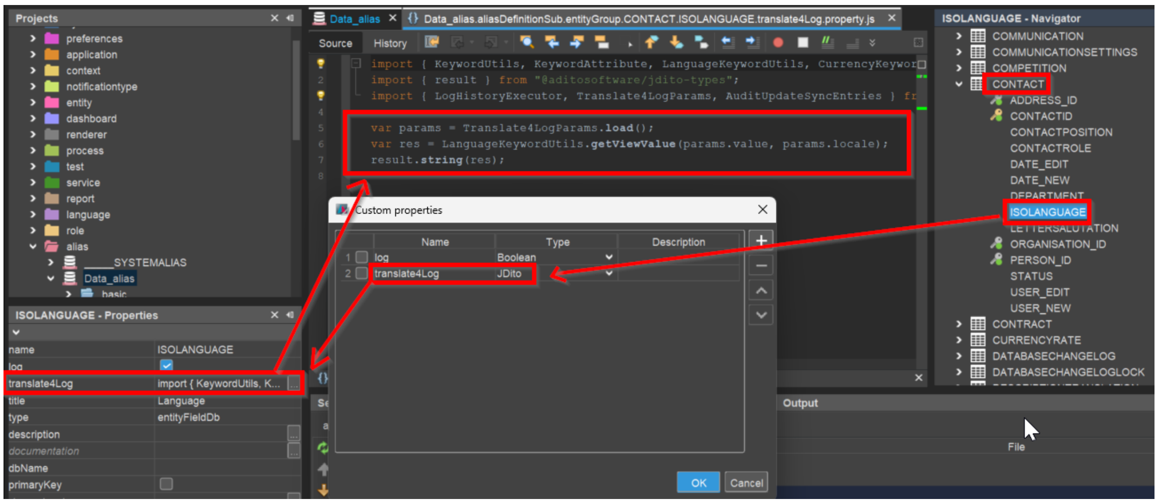
KIND - Properties

name	KIND
log	✓
keyword	OrganisationType
title	Type
type	entityFieldDb
description	
documentation	
dbName	

Custom properties

	Name	Type	Description
1	log	Boolean	
2	keyword	String	

- "translate4Log" (Type: JDito); purpose: resolving displayValues. Example:



Data_alias.aliasDefinitionSub.entityGroup.CONTACT.ISOLANGUAGE.translate4Log.property.js

```
import { KeywordUtils, KeywordAttribute, LanguageKeywordUtils, CurrencyKeywor
import { result } from "@aditosoftware/jdito-types";
import { LogHistoryExecutor, Translate4LogParams, AuditUpdateSyncEntries } fr

var params = Translate4LogParams.load();
var res = LanguageKeywordUtils.getViewValue(params.value, params.locale);
result.string(res);
```

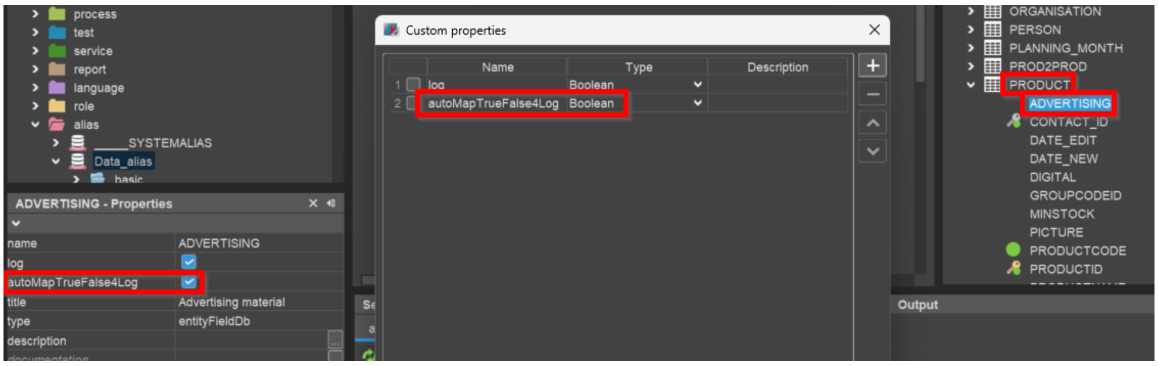
Custom properties

	Name	Type	Description
1	log	Boolean	
2	translate4Log	JDito	

ISOLANGUAGE - Properties

name	ISOLANGUAGE
log	✓
translate4Log	import { KeywordUtils, K...
title	Language
type	entityFieldDb
description	
documentation	
dbName	
primaryKey	

- "autoMapTrueFalse4Log" (Type: Boolean); purpose: resolving booleans without having to do this via translate4Log. Example:



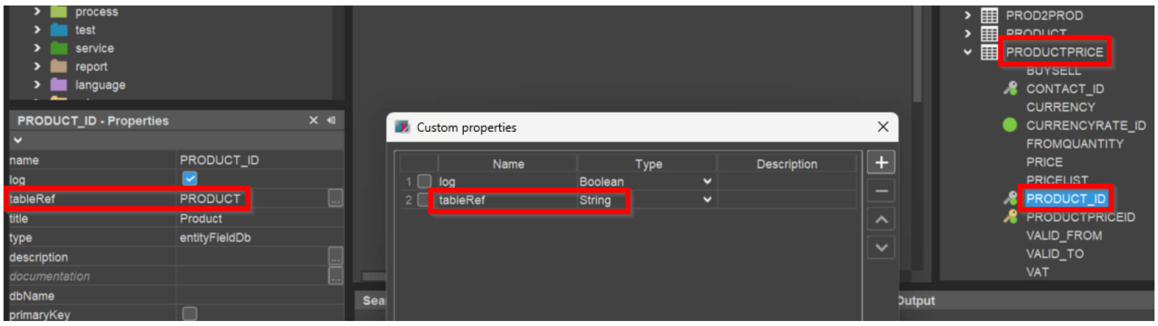
Custom properties

	Name	Type	Description
1	log	Boolean	
2	autoMapTrueFalse4Log	Boolean	

ADVERTISING - Properties

name	ADVERTISING
log	✓
autoMapTrueFalse4Log	✓
title	Advertising material
type	entityFieldDb
description	

- "tableRef" (Type: String); purpose: required in order to log changes in tables to which there is a logging dependency (like in the above example of Organisation_entity and, e.g., COMMUNICATION). Example:



Custom properties

	Name	Type	Description
1	log	Boolean	
2	tableRef	String	

PRODUCT_ID - Properties

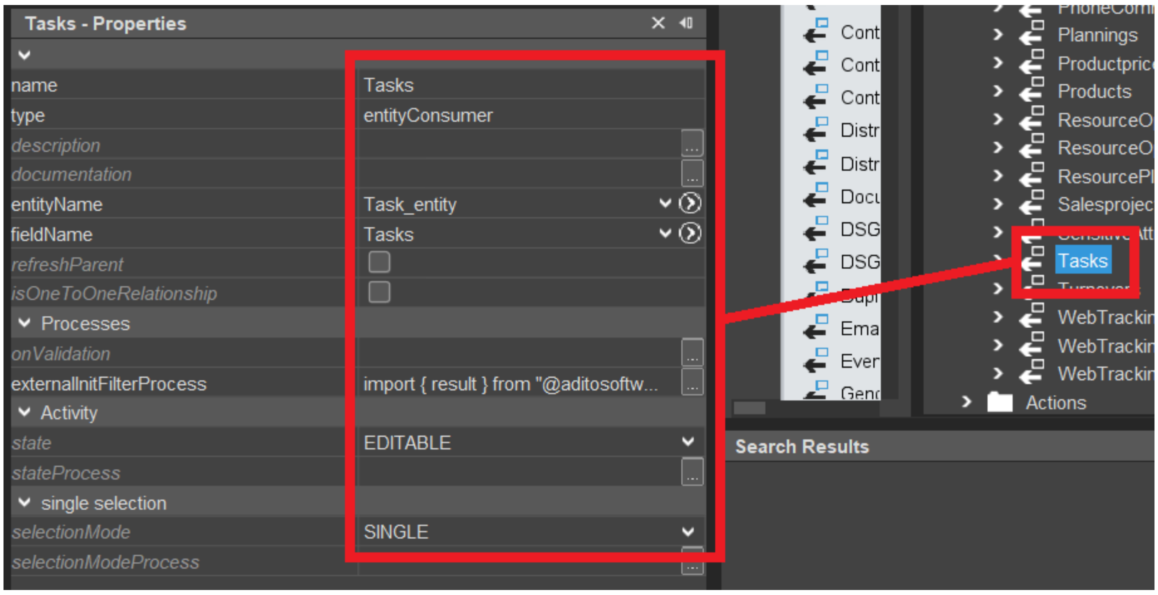
name	PRODUCT_ID
log	✓
tableRef	PRODUCT
title	Product
type	entityFieldDb
description	
documentation	
dbName	
primaryKey	

10.15. Adding Tasks

This chapter explains how to add "Task" functionality to a Context, using the example of Context "Organisation".

These are the required steps:

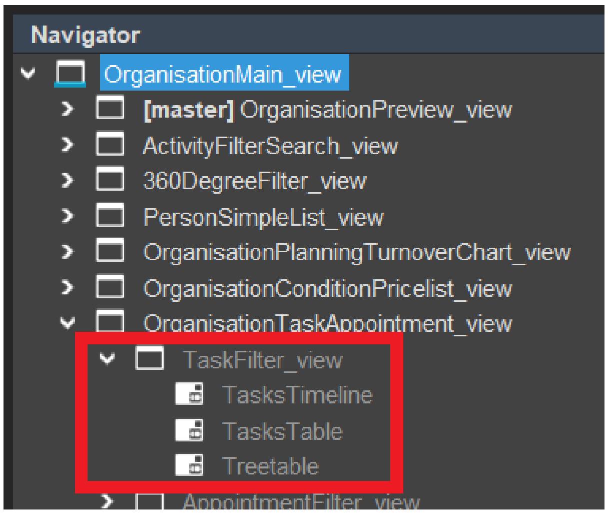
1. Add a Consumer that points to Provider "Tasks" of Task_entity - can be copied from Organisation_entity:



Tasks - Properties

name	Tasks
type	entityConsumer
description	
documentation	
entityName	Task_entity
fieldName	Tasks
refreshParent	☐
isOneToOneRelationship	☐
Processes	
onValidation	
externalInitFilterProcess	import { result } from "@aditosoftw...
Activity	
state	EDITABLE
stateProcess	
single selection	
selectionMode	SINGLE
selectionModeProcess	

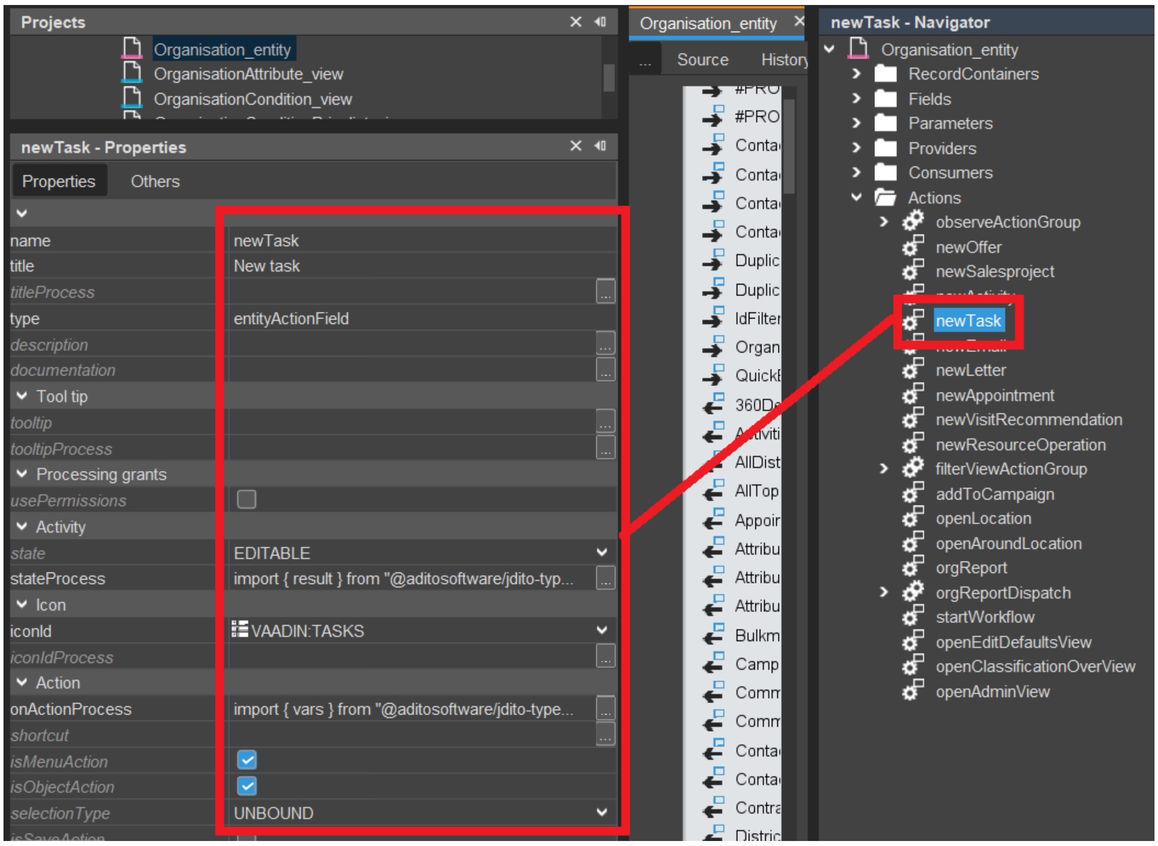
2. In the Context’s MainView, add a reference to TaskFilter_view:



Navigator

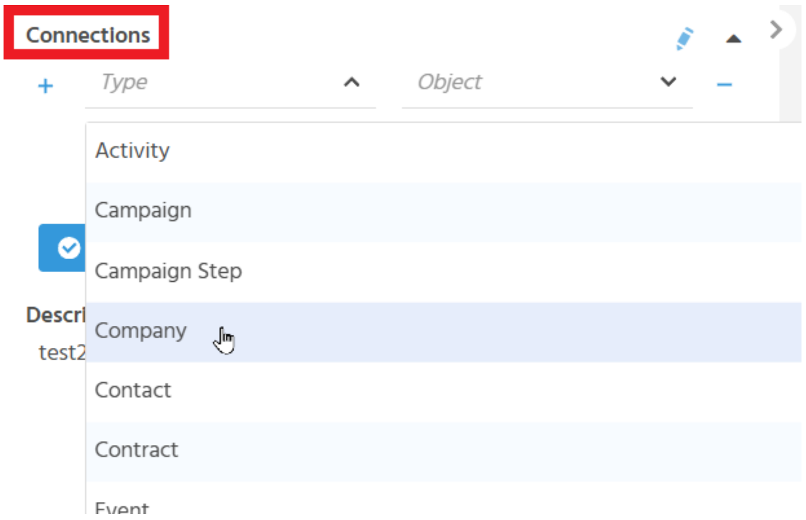
- OrganisationMain_view
  - **[master]** OrganisationPreview_view
  - ActivityFilterSearch_view
  - 360DegreeFilter_view
  - PersonSimpleList_view
  - OrganisationPlanningTurnoverChart_view
  - OrganisationConditionPricelist_view
  - OrganisationTaskAppointment_view
    - TaskFilter_view
      - TasksTimeline
      - TasksTable
      - Treetable
    - AppointmentFilter_view

3. Add Action "newTask" - can be copied from Organisation_entity:



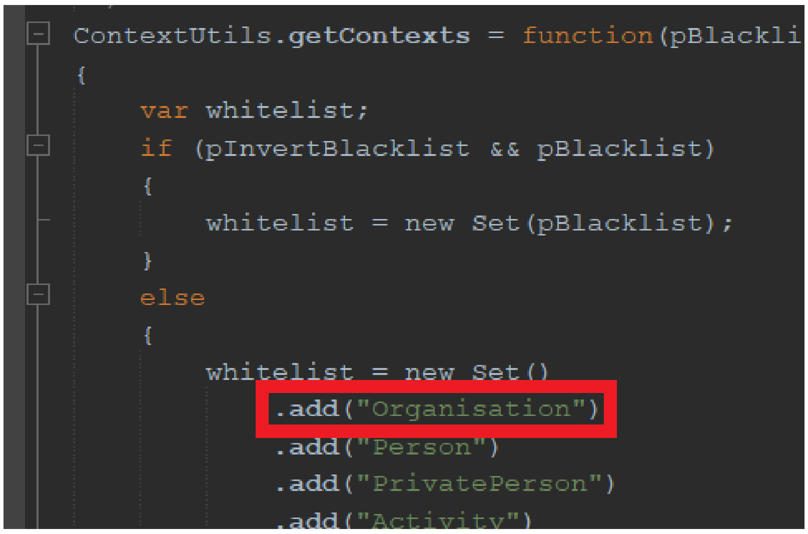
name	newTask
title	New task
titleProcess	
type	entityActionField
description	
documentation	
Tool tip	
tooltip	
tooltipProcess	
Processing grants	
usePermissions	☐
Activity	
state	EDITABLE
stateProcess	import { result } from "@aditosoftware/jdito-typ...
Icon	
iconId	VAADIN:TASKS
iconIdProcess	
Action	
onActionProcess	import { vars } from "@aditosoftware/jdito-type...
shortcut	
isMenuAction	✓
isObjectAction	✓
selectionType	UNBOUND

4. For being available in the selection of the "Connections"-labelled ViewTemplate (of Contexts Tasks, Activities, etc.)…



…the following steps are important:

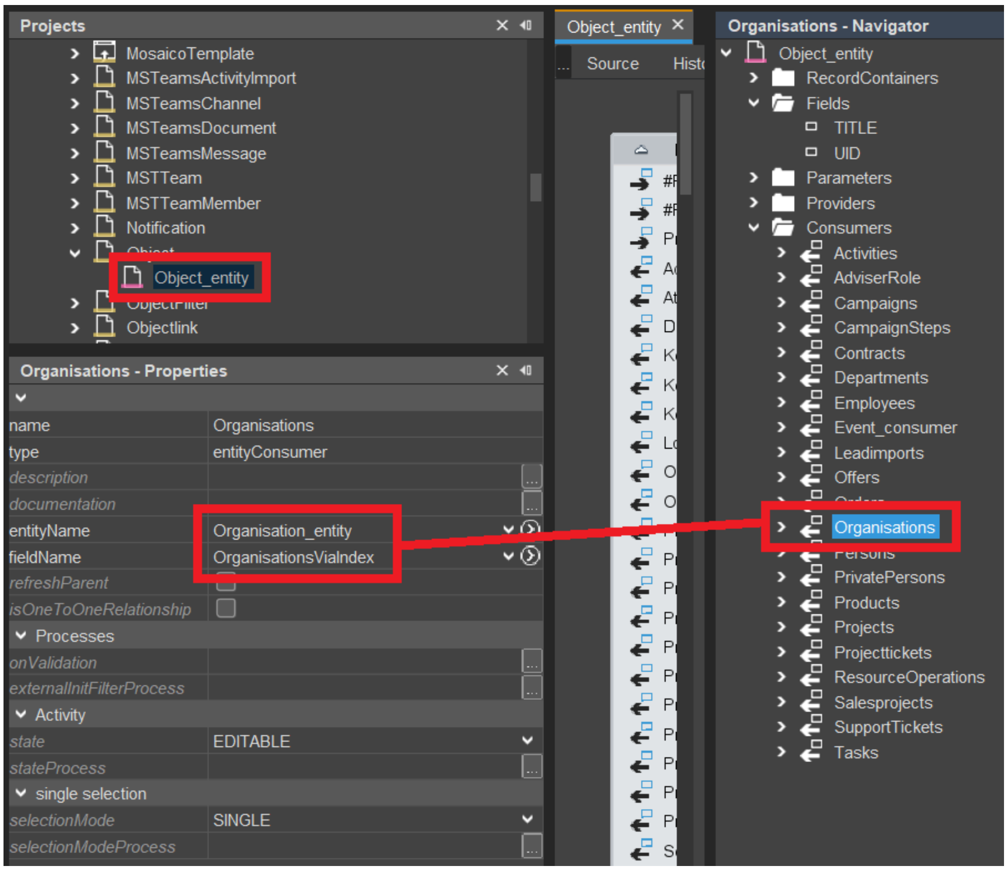
a. `ContextUtils.getContexts(…)` (Context_lib): Add your Context to the whitelist:



```
ContextUtils.getContexts = function(pBlackli
{
    var whitelist;
    if (pInvertBlacklist && pBlacklist)
    {
        whitelist = new Set(pBlacklist);
    }
    else
    {
        whitelist = new Set()
            .add("Organisation")
            .add("Person")
            .add("PrivatePerson")
            .add("Activity")
```

The above applies to unmodularized projects. On the contrary, in modularized projects, you usually work with so-called ServiceImplementations, which are iterated over in function `ContextUtils.getContexts(…)` of Context*Utils*_lib. For further details, see the ADITO Information Document AID123 "Modularization".

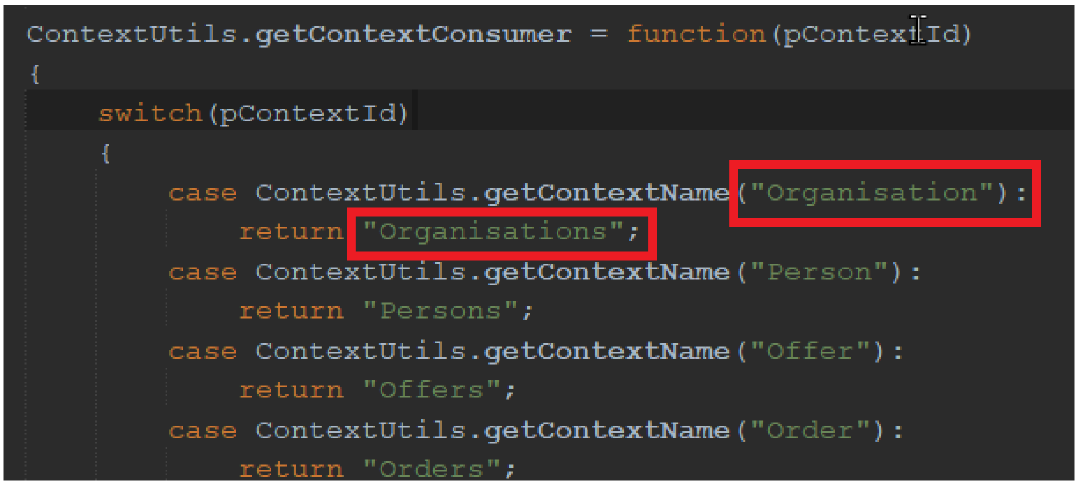
b. Object_entity: Create a Consumer and connect it to a suitable Provider of your Context:



Organisations - Properties

name	Organisations
type	entityConsumer
description	
documentation	
entityName	Organisation_entity
fieldName	OrganisationsViaIndex
refreshParent	☐
isOneToOneRelationship	☐
Processes	
onValidation	
externalInitFilterProcess	
Activity	
state	EDITABLE
stateProcess	
single selection	
selectionMode	SINGLE
selectionModeProcess	

c. `ContextUtils.getContextConsumer` (Context_lib):
Add your Context and the new Consumer (see previous step) of Object_entity (see previous step):



```
ContextUtils.getContextConsumer = function(pContextId)
{
    switch(pContextId)
    {
        case ContextUtils.getContextName("Organisation"):
            return "Organisations";
        case ContextUtils.getContextName("Person"):
            return "Persons";
        case ContextUtils.getContextName("Offer"):
            return "Offers";
        case ContextUtils.getContextName("Order"):
            return "Orders";
```

The above applies to unmodularized projects. On the contrary, in modularized projects, you usually work with so-called ServiceImplementations, which are iterated over in function `ContextUtils.getContextConsumer(…)` of Context*Utils*_lib.

> For further details, see the ADITO Information Document AID123 "Modularization".

10.16. Auto-generated Primary Keys

The following procedure is not required, if you have checked the checkbox "UID Table" in the RecordContainer’s property "linkInformation" (see above). However, you may use it as configuration pattern, if you want to include further fields with an auto-generated UID as content.

For testing purposes, we had inserted the values of the primary keys (e.g., CARID) by ourselves (see Liquibase xml files). In most cases, however, this should be done automatically, whenever a new dataset is generated. To achieve this, we enter some code in the property "valueProcess" of all EntityFields related to the primary key of the database table corresponding to the Entity: CARID, CARDRIVERID, CARRESERVATIONID.

Car_entity.CARID.valueProcess.js, CarDriver_entity.CARDRIVERID.valueProcess.js, CarReservation_entity.CARRESERVATIONID.valueProcess.js

```
if(vars.get("$sys.recordstate") == neon.OPERATINGSTATE_NEW) {
    result.string(util.getNewUUID());
}
```

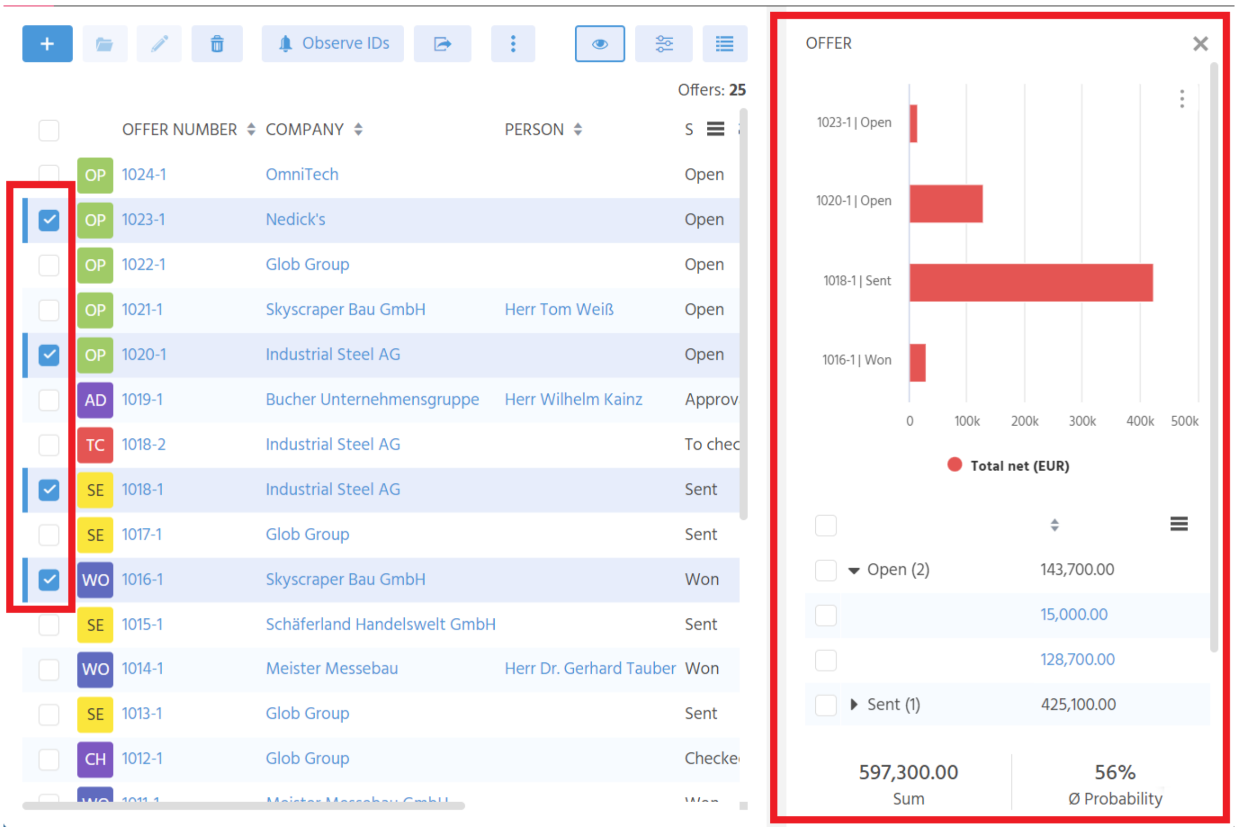
Explanation:

- `result` is similar to what you may know as "return".
- `.string` means that the result value is a text.
- `util` is the library holding utility functionality.
- `getNewUUID()` returns a new randomly generated UID
- The condition `if(vars.get("$sys.recordstate") == neon.OPERATINGSTATE_NEW)` restricts the auto-generation to only the creation mode (i.e., cases when the "plus" sign is clicked). Whenever the system is in another state (VIEW, EDIT), the field simply shows the stored value.

10.17. PreviewMultiple

While the "normal" PreviewView always refers to one single dataset (selected in the FilterView), the PreviewMultiple allows to show processed, summarized, and aggregated data of multiple selected datasets. The respective View is set in the Context’s property "previewMultiple".

Example:
Context "Offer" includes a PreviewMultiple that summarizes the total net and the probability of offer datasets that have been selected in the FilterView:



		OFFER NUMBER	COMPANY	PERSON	S
	OP	1024-1	OmniTech		Open
✓	OP	1023-1	Nedick's		Open
	OP	1022-1	Glob Group		Open
	OP	1021-1	Skyscraper Bau GmbH	Herr Tom Weiß	Open
✓	OP	1020-1	Industrial Steel AG		Open
	AD	1019-1	Bucher Unternehmensgruppe	Herr Wilhelm Kainz	Approv
	TC	1018-2	Industrial Steel AG		To chec
✓	SE	1018-1	Industrial Steel AG		Sent
	SE	1017-1	Glob Group		Sent
✓	WO	1016-1	Skyscraper Bau GmbH		Won
	SE	1015-1	Schäferland Handelswelt GmbH		Sent
	WO	1014-1	Meister Messebau	Herr Dr. Gerhard Tauber	Won
	SE	1013-1	Glob Group		Sent
	CH	1012-1	Glob Group		Checke

Open (2)	143,700.00
	15,000.00
	128,700.00
Sent (1)	425,100.00

597,300.00
Sum

56%
Ø Probability

Configuration:
The Offer Context’s property "previewMultiple" references a View named OfferPreviewMultiple_view. This View has a HeaderFooterLayout and includes 3 ViewTemplates:

- OfferChart: This ViewTemplate shows a bar chart that enables a comparison of the total net of the selected Offer datasets. The label of a bar is the result of Offer_entity’s contentTitleProcess. The configuration of the ViewTemplate is quite simple:
    - type: DynamicMultiDataChart
    - columns: NET
    - chartType: BAR

- StatusTreeTable: This ViewTemplate shows a TreeTable that groups the offers' total net according to status ("open", "sent", "won", etc.), including the sum of all selected offers that have a specific status. Configuration of this ViewTemplate:
    - type: TreeTable
    - columns:
        - entityField: NET
        - aggregateEntityField: NET_aggregate
    - defaultGroupFields: STATUS
    - hideActions: true
- AggregatedValues: This ViewTemplate is set as footer of OfferPreviewMultiple_view. It shows, as "score cards", the sum of the total net of all selected offer datasets as well as the average probability of all selected offer datasets. Configuration:
    - type: ScoreCard
    - entityField: OfferAggregates (= a Provider of Offer_entity, without further configuration)
    - fields: NET_aggregate, PROBABILITY_aggregate

10.18. Paging

From the ADITO client user's point of view, paging means that (e.g., in a table) the datasets of a Context are not loaded all at once, but step by step if you scroll down, in blocks (pages) of n datasets. Usually, the client user notices only a small break when scrolling down, before the next n datasets have been loaded.

From the customizing point of view, the paging approach depends on the RecordContainer you use.

10.18.1. Paging with a DbRecordContainer

Simply navigate to your RecordContainer and set its property "isPageable" to true (checkbox checked). That's all. The data will then be loaded in pages of 400 datasets.

10.18.2. Paging with a JDitoRecordContainer

Normal paging:
(loading the data in blocks of n datasets while the client user scrolls down)

Prerequisites:

- isPageable = true
- isGroupable = false
- rowCountProcess is present

To implement normal paging, you need 2 variables:

- $local.page: returns the number of the page that is requested (0, 1, 2, 3…)
- $local.pagesize: returns how many rows (datasets) are to be loaded per page

A rough, not recommendable implementation of paging would be to finish the contentProcess like this:

```
var page = vars.exists("$local.page") ? vars.get("$local.page") : false;
var pageSize = vars.exists("$local.pagesize") ? vars.get("$local.pagesize") : false;

if(pageSize)
{
var startRow = page == false ? 0 : page * pageSize;
var endRow = startRow + pageSize;
res = res .slice(startRow, endRow)
}
```

Then only the requested rows are returned, but still all rows are loaded. Thus, this kind of "paging" does not have an added value. The better way is to load only the requested datasets, using the above mentioned 2 variables.

Paging on basis of grouping:
(on the basis of grouping, the datasets are grouped tree-wise)

- isPageable = true
- isGroupable = true
- rowCountProcess is present

You can learn how to implement this kind of paging by looking at the RecordContainers of, e.g., the Contexts

- SalesprojectConversionRate
- Turnover

As soon as grouping is applied, variable $local.grouped is set, and you do not need to return the data, but the respective groups. If you open a group and there are no sub-groups below, then the respective data is to be returned. The filter of the grouping(s) is then included in variable $local.filters.

10.18.3. Further information

For any RecordContainer, you need to decide if you use caching or paging. It is not possible to use both at the same time.

10.19. Storing user-specific data outside ASYS_USERS

User-specific data such as title, configuration, etc. is stored in the system table "ASYS_USERS". As this table is already very large, it should not be further extended by customizing.

Instead, if you intend to store further user-specific data, it is advisable to store this data in a separate table within your Data_alias (not the system alias!). This approach improves performance by reducing the amount of data that needs to be loaded from ASYS_USERS.

The prerequisite for this new table is the use of the OBJECT_ID and OBJECT_TYPE columns. An illustrative example is the Entity TopicTreeTopicConfiguration_entity with its corresponding database table TOPICTREETOPICCONFIGURATION.

10.20. Lookup for translated values

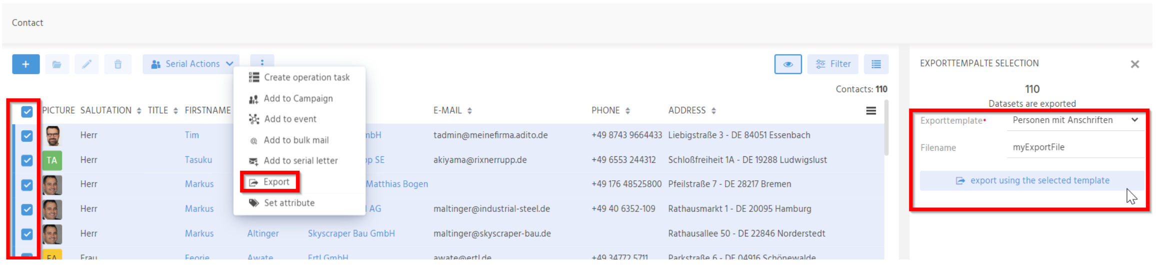
In an international context, lookup functionality is usually refering to translated search content. How this can be realized, depends on the type of RecordContainer:

- DBRecordContainer: Use the SQL expression for the displayValue as Filter. However, depending on the SQL, this could be a performance issue.
- JDitoRecordContainer: Can be filled via the contentProcess

10.21. Export

As you can see, e.g., in the FilterView of Context Person, there is an Action named "Export", which enables you to export specific fields of specific datasets. This is how it works:

1. Open PersonFilter_view.
2. Mark a few or all datasets.
3. Execute Action "Export" in ActionGroup "Serial Actions".
4. Choose an export template, e.g. "Persons with addresses".
5. Enter a filename of your choice for the csv export file to be generated.
6. Click button "export using the selected template".



PICTURE	SALUTATION	TITLE	FIRSTNAME			E-MAIL	PHONE	ADDRESS
	Herr		Tim		nbH	tadmin@meinefirma.adito.de	+49 8743 9664433	Liebigstraße 3 - DE 84051 Essenbach
TA	Herr		Tasuku		p SE	akiyama@rixnerrupp.de	+49 6553 244312	Schloßfreiheit 1A - DE 19288 Ludwigslust
	Herr		Markus		Matthias Bogen		+49 176 48525800	Pfeilstraße 7 - DE 28217 Bremen
	Herr		Markus		AG	maltinger@industrial-steel.de	+49 40 6352-109	Rathausmarkt 1 - DE 20095 Hamburg
	Herr		Markus	Altinger	Skyscraper Bau GmbH	maltinger@skyscraper-bau.de		Rathausallee 50 - DE 22846 Norderstedt
EA	Frau		Feorie	Awate	Ertl GmbH	awate@ertl.de	+49 34772 5711	Parkstraße 6 - DE 04916 Schönewalde

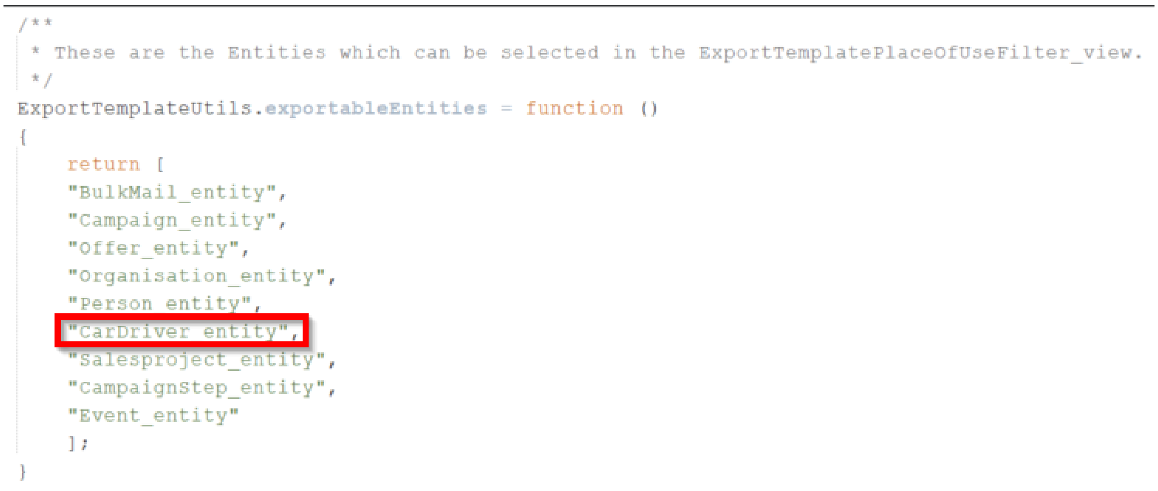
Exporttemplate• Personen mit Anschriften
Filename myExportFile

7. Navigate to your notifications (bell icon).
8. Click on notification "Download ready".
9. Navigate to the PreviewView and click button with download icon.
10. Find your export file in your download folder.

Now we want to implement this export feature also for our Context CarDriver. Please proceed as follows:

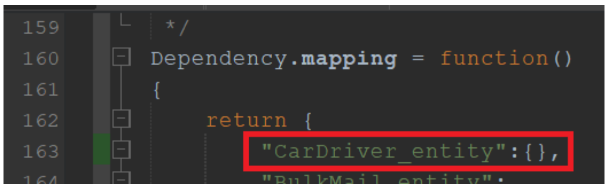
- Set CarDriver_entity's properties "useFavorites" and "recordsRecipeSupported" both to "true" (checkbox checked)
- Copy action "Export" from Person_entity to CarDriver_entity. This requires the following steps:
  - Create an ActionGroup in CarDriver_entity and name it, e.g., "FilterViewActions".
  - Navigate to Person_entity and unfold ActionGroup "filterViewActionGroup". Here, right-click on Action "export" and choose option "Copy".
  - Navigate back to CarDriver_entity and right-click on ActionGroup "FilterViewActions". From the context menu, choose option "Paste". This will copy Action "export" with all its properties, in particular, onActionProcess (containing the export logic) and stateProcess (making sure that the Action is only enabled, if at least one dataset is marked).

- Set the copied ActionGroup in one of the properties "FavouriteActionGroupX" in the TableViewTemplate assigned to CarDriverFilter_view.
- Open ExportTemplate_lib (in the project tree, under process > libraries) and add "CarDriver_entity" in function `ExportTemplateUtils.exportableEntities`:



```
/**
 * These are the Entities which can be selected in the ExportTemplatePlaceOfUseFilter_view.
 */
ExportTemplateUtils.exportableEntities = function ()
{
    return [
    "BulkMail_entity",
    "Campaign_entity",
    "Offer_entity",
    "Organisation_entity",
    "Person_entity",
    "CarDriver_entity",
    "Salesproject_entity",
    "CampaignStep_entity",
    "Event_entity"
    ];
}
```

- Open Dependency_lib (in the project tree, under process > libraries) and add an empty entry for "CarDriver_entity" in function `Dependency.mapping`:



```
 */
Dependency.mapping = function()
{
    return {
        "CarDriver_entity":{},
        "BulkMail_entity":
```

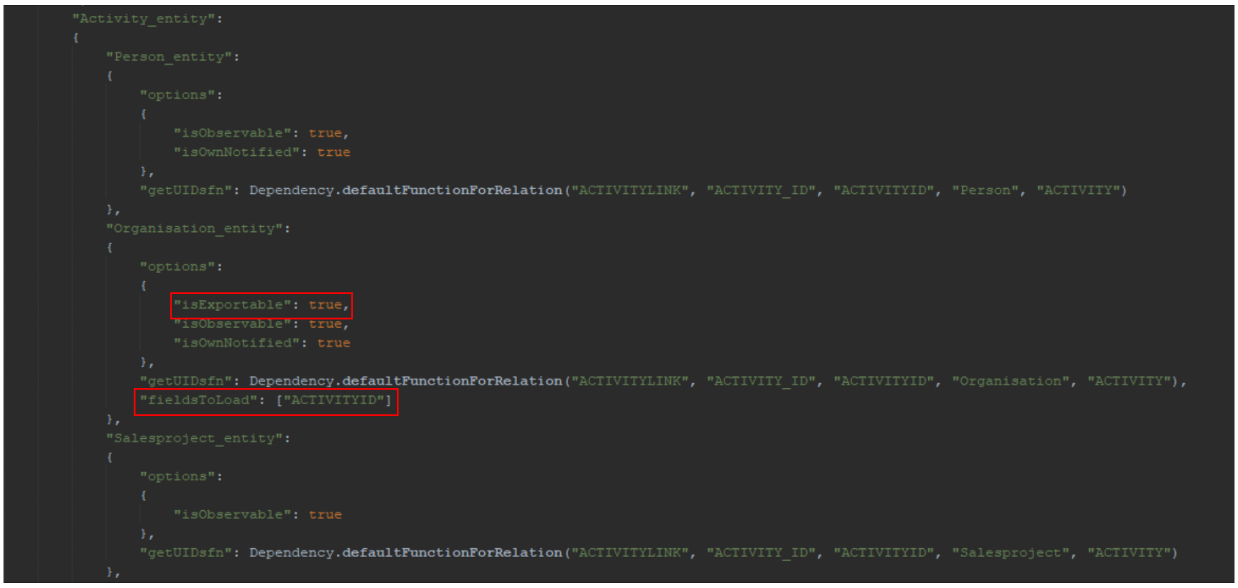
- Deploy
- In the client, navigate to Context "Export Template" (in menu group "Administration").
- Click the "Plus" button in order to add a new template. Give it a suitable name and description and select "Car driver" as "Place of use". Change the other options according to your requirements.
- Choose "Save and open entry"
- Choose tab "Export Template Fields". Click the "Plus" button, choose "Car driver" as "Place of use" and then choose the first export field.
- Continue with the second, third, etc. export field.
- If required, change the order via the "arrow up/down" buttons.

Now you can test the added export functionality by proceeding as described above for Context Person. Furthermore, you can test your skills by adding the same export functionality to the Contexts Car and CarReservation.

10.21.1. Export of a subordinated Entity

If you want to export a subordinated Entity in an export template, you need to modify the configuration of the subordinated Entity in library Dependency_lib, method `Dependency.mapping`.

Example: Activity_entity, with its subordinated Entity Organization_entity. In this case, the nodes `isExportable` and `fieldsToLoad` need to be added as follows:



```
"Activity_entity":
{
    "Person_entity":
    {
        "options":
        {
            "isObservable": true,
            "isOwnNotified": true
        },
        "getUIDsfn": Dependency.defaultFunctionForRelation("ACTIVITYLINK", "ACTIVITY_ID", "ACTIVITYID", "Person", "ACTIVITY")
    },
    "Organisation_entity":
    {
        "options":
        {
            "isExportable": true,
            "isObservable": true,
            "isOwnNotified": true
        },
        "getUIDsfn": Dependency.defaultFunctionForRelation("ACTIVITYLINK", "ACTIVITY_ID", "ACTIVITYID", "Organisation", "ACTIVITY"),
        "fieldsToLoad": ["ACTIVITYID"]
    },
    "Salesproject_entity":
    {
        "options":
        {
            "isObservable": true
        },
        "getUIDsfn": Dependency.defaultFunctionForRelation("ACTIVITYLINK", "ACTIVITY_ID", "ACTIVITYID", "Salesproject", "ACTIVITY")
    },
```

Figure 31. Dependency_lib.Dependency.mapping

11. Controlling the design

In order to achieve a professional appearance and an ergonomic handling of every ADITO application, the possibilities to control its design (colors, order of elements, etc.) are basically reduced to three complementary options: Themes, ViewTemplates, and layouts:

- A so-called **Theme** contains configurations affecting all Views and Contexts, particularly regarding colors.
- A **ViewTemplate** defines what and how data is presented, e.g., specific fields of one dataset ordered in single lines, or multiple datasets in table form, with or without Action buttons, etc.
- A **layout** determines the way multiple associated ViewTemplates are presented together in a View (horizontally or vertically ordered, selectable via a button, etc.)

We recommend you to respect the ADITO Design Guideline for all aspects of your customizing work. Find more information in the ADITO Information Document (AID) "AID003 Design Guideline", which is available in the customer area of the ADITO web site.

Now let’s have a closer look at each of these options and their variations.

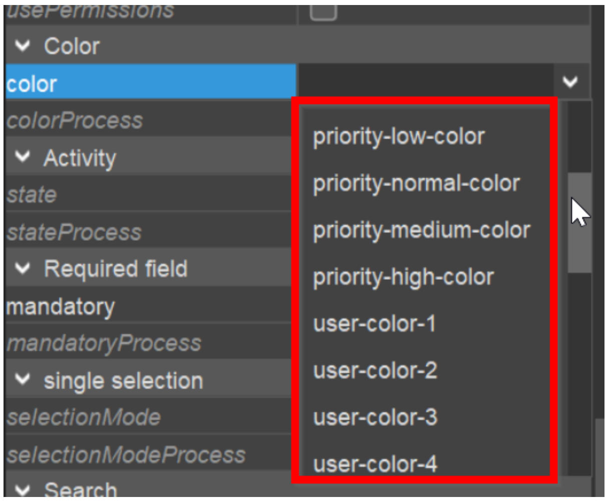
11.1. Themes

A Theme contains design configurations at a very fundamental level. The settings included in a Theme affect all display components of an ADITO application, particularly Views and Contexts. Currently, a Theme is limited to the definition of colors of various visual elements in the ADITO client.

As color design is quite complex and has to consider critical aspects like contrast, Themes can currently not be created or customized by the user, but exclusively be ordered from ADITO, on the basis of the customer's CI guide. This will ensure that a Theme is in good compliance with the customer's company colors.

Once created, a new Theme can easily be integrated into an existing ADITO application: All the ADITO administrator has to do, is to place a specific configuration file (supplied by ADITO's development department) in a specific folder of the server's file structure and restart the server. Then, the name of the new Theme can be selected in the Designer: system > default > _CONFIGURATION > System > Client > clientTheme. By default, this property is not set. Then, the ADITO standard Theme will be used, which is included in every ADITO installation.

You can view the available colors defined by the Theme if, e.g., you open the combo box of any "color" property (e.g., the color property of an EntityField).



It is strictly against the intention of ADITO that users modify the Theme by themselves. It is **exclusively** ADITO's development department that is authorized to modify a Theme or create a new Theme. In appendix Requirements for customized Theme you can find information about the information required by the ADITO development department in order to supply you with a customized Theme. Furthermore, you can find extensive background information on the topic "Themes" in the ADITO Information Document AID121 "Themes".

11.2. Layouts

A layout is a property of a View. It determines the way multiple ViewTemplates (or referenced Views) are presented together in a View, e.g., whether they are horizontally or vertically ordered, selectable via a button, etc. The ADITO standard installation includes a set of predefined layouts, which fit for most use cases.

In order to assure a consistent and ergonomic design, layouts cannot be created or customized by the user. In special cases, if none of the predefined layouts fits, a customized layout can be ordered from ADITO.

To set a layout, open a View in the "Projects" windows or Navigator window, click on it and edit property "layout" in the "Properties" window. Here, you can select a layout from a list of layouts, in a combo box. Depending on what layout you select, different layout-specific properties are shown below the "layout" property.

Below, the function and configuration of selected layouts is explained.

11.2.1. NoneLayout

The NoneLayout is the simplest layout. It shows all ViewTemplates assigned to the View in a vertical arrangement. The order of the ViewTemplates is the same as shown in the Navigator window.

This layout is often used, whenever only one single ViewTemplate or one single View is to be displayed.

Example:
KeywordAttributeEdit_view (Context "KeywordAttribute"). Visible in the client, whenever you create or edit a keyword attribute (Administration > Keyword Attribute).

11.2.2. DrawerLayout

The DrawerLayout is almost as simple as the NoneLayout, with the following differences:

- A horizontal bar is shown on top. Optionally, this bar can include a caption (property "layoutCaption").
- Via a button at the right end of the bar (icon "^") the user can hide all Views and ViewTemplates assigned to the View having the DrawerLayout.
- Property "fixedDrawer": If checked, the above "hide" function is disabled. Then, the horizontal header bar and its optional caption is the only difference to the NoneLayout.

Example:
AppointmentFilter_view (Context "Person", assigned to PersonTaskAppointment_view, which in turn is assigned to PersonMain_view). Visible in the client under Contact Management > Contact: Select any

person and press the "open" button, to open the MainView. Here, click on tab "Tasks": In the lower part of this tab, you can see a table, above which there is a horizontal bar, including the caption "Linked Appointments". Via a button at the right end of the bar (icon "^") you can hide the table.

11.2.3. BoxLayout

The BoxLayout is also a quite basic layout, but it has some more properties than the NoneLayout:

- direction: Select, whether the ViewTemplates are to be shown in vertical or in horizontal order.
- maxDirectionElements: Specify the maximum of ViewTemplates to be shown in the order specified in property "direction". If, e.g., direction is VERTICAL, and maxDirectionElements is 3, then the first 3 ViewTemplates are shown one under the other, while the 4th ViewTemplate is shown on top again, to the right of the first ViewTemplate.
- autoHeight: Check, if the layout should determine its height automatically. The automatic layout height is calculated from the height of its components.

Example:
OrganisationEdit_view (Context "Organisation"). Visible in the client under Contact Management > Company, whenever you create or edit a company dataset. You see that, as configured, the Views showing address data, communication data, and attribute data, are shown in vertical order.

For every layout, you can change the order of the ViewTemplates simply by dragging and dropping them in the Navigator window up or down.

11.2.4. GroupLayout

The GroupLayout enables you to change the visualization between multiple assigned Views or ViewTemplates via a button shown in the upper right corner. If you click on this button, a list of all Views is shown, which are assigned to the View having the GroupLayout. As soon as you have selected a View, this View is shown, and all other Views are hidden.

Example:
ActivityFilter_view (Context "Organisation", assigned to OrganisationMain_view). Visible in the client under Contact Management > Company: Select any company and press the "open" button, to open the MainView. Here, click on tab "Activities" to show all activities of the person. Via the button on the right, you can change the visualization of the activities: You can switch between 3 ViewTemplates: Timeline View, Table View, and Treetable View.

To customize the naming of a list item shown via the View selection button, you can set an arbitrary text in the "title" property of the respective ViewTemplate. Unless you set the "title" property, a (non-configurable) default name is shown.

11.2.5. HeaderFooterLayout

The HeaderFooterLayout divides the View into 3 parts:

- header area: Upper part, which can consist of a View or a ViewTemplate.
- footer area: Lower part, which can consist of a View or a ViewTemplate.
- middle area: All other Views or ViewTemplates assigned to the View having the HeaderFooterLayout.

In many cases, the HeaderFooterLayout is used for the PreviewView.

To configure this layout, proceed as follows:

1. Create and select the View that should get the HeaderFooterLayout.
2. Select "HeaderFooterLayout" from the combo box of property "layout".
3. Open the View in the Navigator window.
4. Create all ViewTemplates to be displayed in the View (context menu: "New ViewTemplate…").
5. Assign all other Views to be displayed in the View (context menu: "Add reference to existing View…").
6. Select the View having the HeaderFooterLayout and edit its properties:
   a. Property "header": Select the View or ViewTemplate that is to be shown in the header area.
      == .. Property "footer": Select the View or ViewTemplate that is to be shown in the footer area.
7. Open the View having the HeaderFooterLayout in the Navigator window.
8. Configure the order in which the Views and ViewTemplates are to be displayed: Move all Views and ViewTemplates up or down, until they are in the required order, simply by dragging and dropping them. This is only necessary for the Views and ViewTemplates to be shown in the middle area. (The ViewTemplates assigned as "Header" or "Footer" have fixed positions.)

Example:
OrganisationPreview_view (Context "Organisation"). Visible in the client under Contact Management > Company: Click on any company and watch its data displayed in the preview on the right.

11.2.6. GridLayout

The GridLayout enables you to place ViewTemplates (or View references, respectively) in a grid. The placement of the ViewTemplates is determined by their order as given in the Designer’s Navigator window. They get filled into the grid line by line, from top to bottom.

The grid is defined by the number of its columns (property "columnCount") and its rows (property "rowCount"). Optionally, property "rowHeight" can be set. In most cases, only columnCount needs to be set, while rowCount and rowHeight are left in default state. Then, the number of rows is calculated automatically, and rowHeight is automatically optimized.

> In some ADITO versions, property "rowHeight" will not appear until you have changed the value of property "columnCount" at least once. Furthermore, the automatic calculation of the properties "rowCount" and "rowHeight" might fail. In these cases, set "columnCount" and "rowCount" (and, if required, "rowHeight") explicitely.

11.2.6.1. Properties

- **columnCount**
  Determines how many columns the grid should have.

  > This value is only used for desktop devices. Mobile devices will always only use **1** column.

- **rowCount**
  Determines the number of rows which should be displayed in the grid. If left in default state (recommended), this property gets adapted automatically when you add/remove ViewTemplates to/from the View.

  > If you explicitly set this property, you should also consider to set property rowHeight (see below).

- **rowHeight**
  The height of one row in pixels.

  > This property should only be used in combination with an explicitly set rowCount.

11.2.7. MasterDetailLayout

See chapter "MasterDetailLayout", subchapter of chapter "Complex dependencies".

> Note that DashletConfigs cannot be added to Views having a MasterDetailLayout.

11.3. ViewTemplates

A ViewTemplate defines what and how data is presented, e.g., specific fields of one dataset ordered in single lines, or multiple datasets in table form, with or without Action buttons, etc. The ADITO standard installation includes a set of predefined ViewTemplates, which fit for most use cases.

In order to assure a consistent and ergonomic design, ViewTemplates cannot be created or customized by the user. In special cases, if none of the predefined ViewTemplate types fits, and a workaround is not possible, then a customized ViewTemplate can be ordered from ADITO.

To add a ViewTemplate to a View, open the View in the Navigator window and choose "Add ViewTemplate" from its context menu. Consequently, a dialog will open, in which you select one of the ViewTemplate types included in the list on the left. Furthermore, enter a name for the new ViewTemplate. The field "Assign to" is only required for configuring a MasterDetailLayout and a HeaderFooterLayout (see chapter on layouts); in other cases, it can be ignored.

After creating a new ViewTemplate, it appears in the Navigator window as sub-node of the View it refers to. If you click on the ViewTemplate, you can edit its properties in the "Properties" window. The following properties are common to multiple or all ViewTemplates:

- title: The text of the list item shown via the View selection button, if the ViewTemplate is intergrated into a GroupLayout. (Unless you set the "title" property, a (non-configurable) default name is shown.)
- maxDBRow: Allows you to set a limit for the number of datasets to be displayed in the ViewTemplate. CAUTION: Depending on your ADITO version, all datasets exceeding this number will be ignored without notification (e.g., when using the filter bar above a ViewTemplate of type TreeTable).
- entityField: The name of the EntityField that should be available for loading. Setting this property to "#ENTITY" means that *all* fields of the Entity can be loaded and are therefore available in the EntityField-related properties, e.g., "columns" or "fields". If you actually need only one single EntityField (e.g., in ViewTemplate "WebContent"), you should select it in property "entityField" accordingly, because this will restrict the loading process and therefore result in a better performance.

Below, the function and configuration of all available ViewTemplate types is explained (in alphabetical order), with the prerequisite that property entityField is set to "#ENTITY".

11.3.1. ActionList

A ViewTemplate of type "ActionList" is used to show a vertical list of 2 fields (icon, title) of multiple datasets, with the title having a hyperlink to execute an Action.

Example:
"Actions", a ViewTemplate of DocumentList_view, being referenced in DocumentTemplatePreview_view (Context "DocumentTemplate").

Appearance in the client:
In the client, you can, e.g., find it under Marketing > Document Template > Click on any document template, then you see the ActionList in the PreviewView, under "MAINDOCUMENTS". The lists consists of the documents' names. If you click on a document, it is downloaded.

Configuration:

"Actions" has the following fields of Document_entity specified: NAME (titleField), DESCRIPTION (descriptionField; visible via a click on the little "eye" icon), ICON (iconField). The specified Action is "downloadSingleFileAction".

11.3.2. Actions

Displays an area with buttons, each related to an Action of a specific Entity. The property "fields" is deprecated and should not be used anymore.

11.3.3. Card

A ViewTemplate of type "Card" displays up to 5 EntityFields of one single dataset, styled like a business card. All fields have fixed positions: On the left, an image (property "iconField"), on the right, in vertical order, 4 further, fields (properties "titleField", "subtitleField", "descriptionField", and "informationField"), of which only the latter one has a label. The Action buttons are shown between descriptionField and informationField.

Optionally, you can add up to 2 Action buttons by selecting an Action in field "favoriteAction1" or "favoriteAction2".

This ViewTemplate is commonly used as "Header" of a HeaderFooterLayout (see chapter Layouts).

Example:
"Head", a ViewTemplate of SalesprojectPreview_view (Context "Salesproject").

Appearance in the client:
In the client, you can find it under Sales > Salesproject. Make sure that the "Preview" button (eye icon) is active (= framed blue). If you click on a project in the table of the Filter View, you see the "Card" ViewTemplate as "Head" (on top) of the PreviewView: It shows an image (or its placeholder) on the left, as well as project title, company name, and project code on the right, followed by Action buttons,

including "New Activity".

In this case, the informationField has been left empty. If set, it appears below the Action buttons, preceded by a label.

Configuration:
"Head" has 4 fields of Salesproject_entity specified: IMAGE (iconField), PROJECTTITLE (titleField), CONTACT_ID (subtitleField), and PROJECTCODE (descriptionField). You may, for testing purposes, set property informationField to STATE, in order to see the effect in the client. Furthermore, property favoriteAction1 is set to newActivity.

11.3.4. CardTable

Displays multiple datasets, each styled like a business card, in a vertical list. Layout and configuration of every card is similar to that of template type "Card".

11.3.5. DragAndDrop

A ViewTemplate of type "DragAndDrop" displays an area including a file selection component and the ability to process files dropped here. As soon as a file is selected or dropped, an Action (property "dropAction") is executed. Additionally, a description can be displayed in the drop zone ("descriptionField").

Example:
"Dropzone", a ViewTemplate of UniversalFileProcessorDropzone_view (Context "UniversalFileProcessor").

Appearance in the client:
In the client, you can, e.g., find it under Marketing > Document Template > Click on the blue "plus" button to create a new document template. You can find on the top of the create form.

Configuration:
"Dropzone" has only one field of UniversalFileProcessor_entity set, namely INFO (descriptionField). As drop Action, the Action "drop_action" is specified.

11.3.6. DynamicForm

A ViewTemplate of type "DynamicFormViewTemplate" enables you to create dynamic forms. The fields of the form do not require EntityFields, but they are generated dynamically, based on a JSON field definition.

Example:
"DynamicForm", a ViewTemplate of WorflowTaskForm_view.

Appearance in the client:
In the client, you can find it in all Contexts that enable the user to create a WorkflowTask: There is a Dashlet for WorkflowTasks. In the WorkflowTaskPreview_view, you can show the form (for an active task), enter data, and finalize the task.

Configuration:

- "formDefinition": Here, you specify an EntityField supplying the JSON string that holds the field definition (see below). The data source of this EntityField is often a JDitoRecordContainer.
- "formResult": When the form is saved, the results will be set to a second EntityField, to be specified in property "formResult" (also in JSON format). The data source of this EntityField is often a JDitoRecordContainer.
- This ViewTemplate can use the following content types:
  - TEXT
  - NUMBER
  - DATE
  - BOOLEAN

The JSON field definition consists of a list of form objects. All form objects have the same data structure:

Configuration pattern for form object of JSON field definition

```
{
    "type": "object",
    "properties": {
       "id":        { "type": "string" },
       "name":              { "type": "string" },
       "contentType":       { "type": "string" },
       "isReadable":        { "type": "boolean" },
       "isWritable":        { "type": "boolean" },
       "isRequired":        { "type": "boolean" },
       "value":         { "type": "string" },
       "possibleItems":         { "type": "object" } (key(string) : value(string))
}
```

Example code for form object of JSON field definition

```
(...)
{
    "id": "propId",
    "name": "propName",
    "contentType": "TEXT",
    "isReadable": true,
    "isWritable": true,
    "isRequired": false,
    "possibleItems": {
    "value1": "Value 1",
    "value2": "Value 2"
    }
}
(...)
```

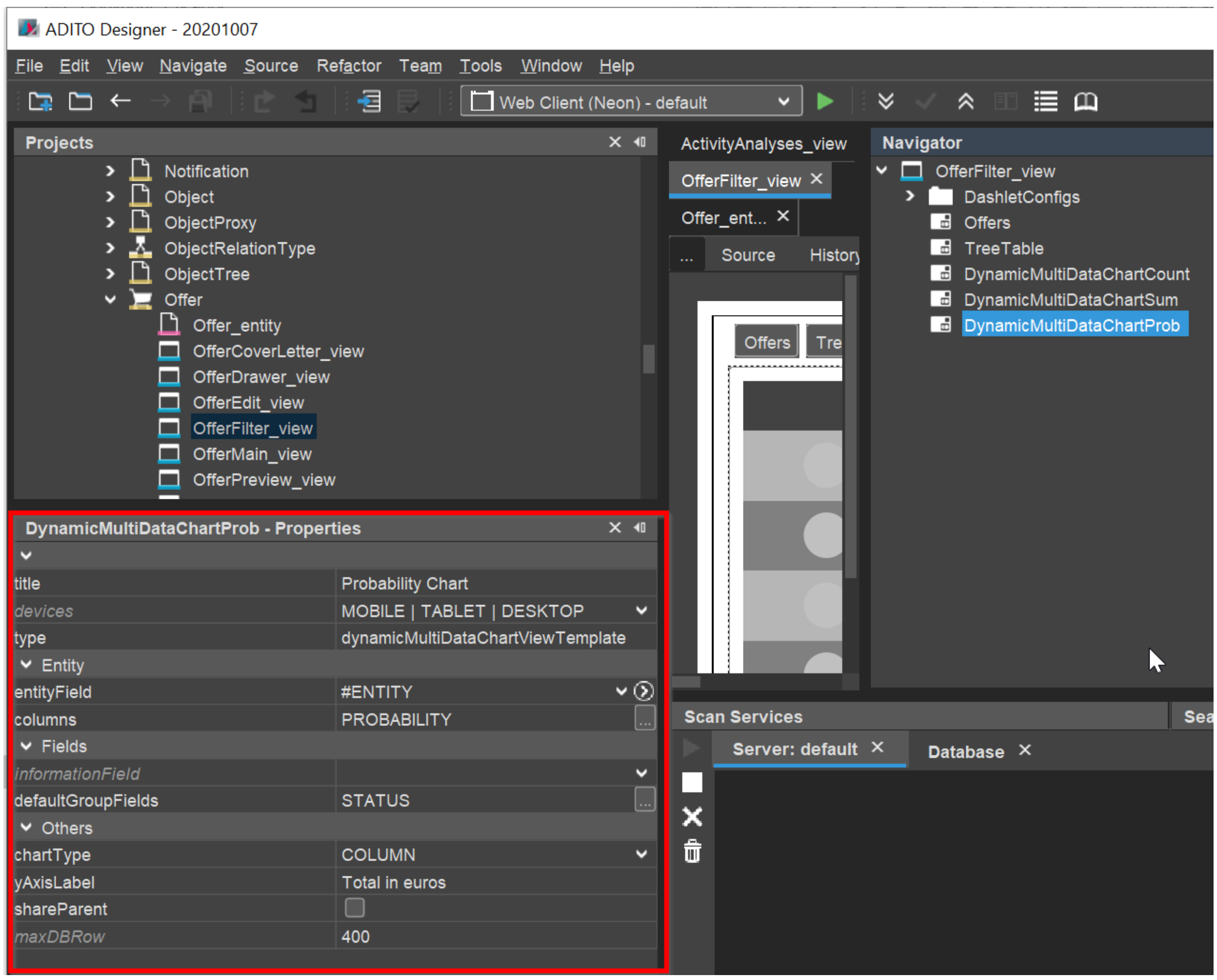
11.3.7. DynamicMultiDataChart

A ViewTemplate of type "DynamicMultiDataChart" is visually similar to the ViewTemplate "MultiDataChart". It displays a multi-dimensional chart with simplified configuration. Only the EntityField and the corresponding AggregateField need to be defined. Client users can create their own chart by grouping and selecting the y-axis.

Example:
"DynamicMultiDataChartProb", a ViewTemplate of OfferFilter_view.

Appearance in the client:
In the client, you can find it under Sales > Offer > Probability Chart (select it in the dropdown menu of the button related to this FilterView's GroupLayout). Probability Chart shows several labeled columns, e.g., "Checked", "Open", and "Sent".

Configuration:



DynamicMultiDataChartProb - Properties

title	Probability Chart
devices	MOBILE | TABLET | DESKTOP
type	dynamicMultiDataChartViewTemplate
Entity	
entityField	#ENTITY
columns	PROBABILITY
Fields	
informationField	
defaultGroupFields	STATUS
Others	
chartType	COLUMN
yAxisLabel	Total in euros
shareParent	☐
maxDBRow	400

11.3.8. DynamicSingleDataChart

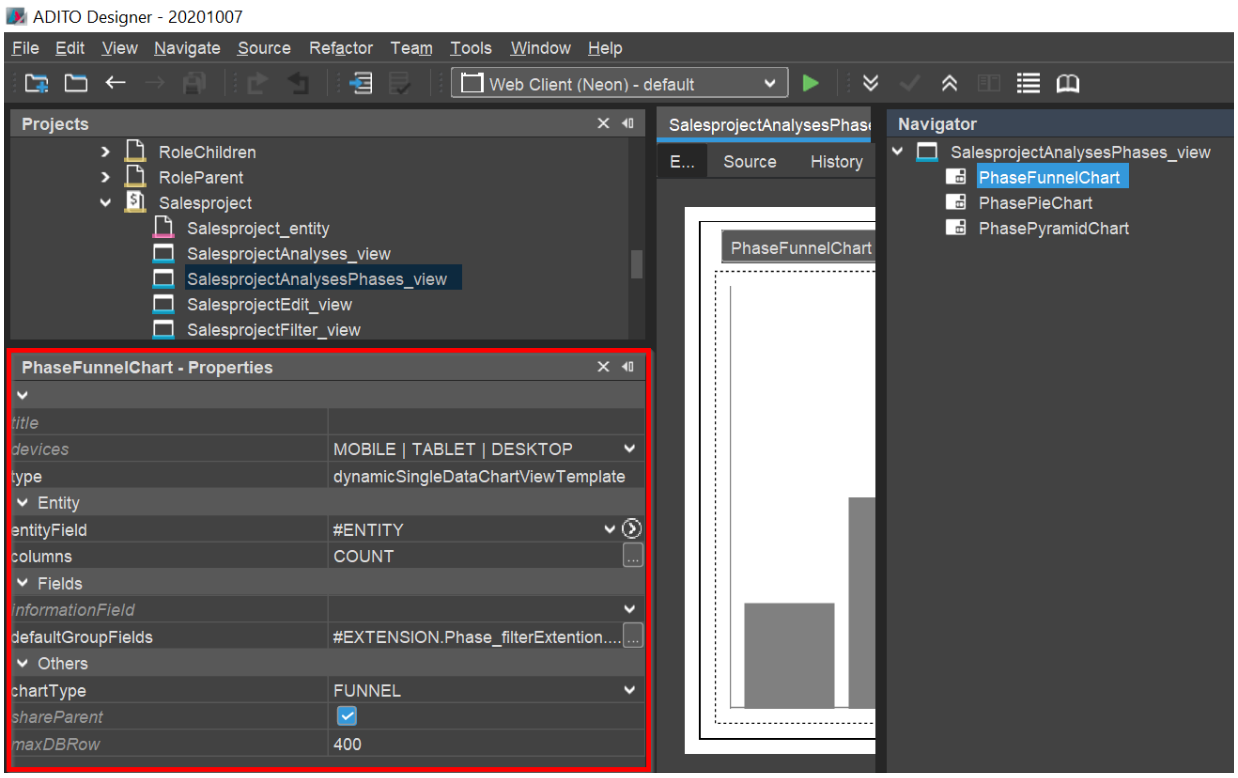
A ViewTemplate of type "DynamicSingleDataChart" is visually similar to the ViewTemplate "SingleDataChart". It displays a single-dimensional chart with simplified configuration. Only the EntityField and the corresponding AggregateField need to be defined. Client users can create their own chart by grouping and selecting the y-axis.

Examples:
"PhaseFunnelChart", "PhasePieChart", and "PhasePyramidChart" - all of them are ViewTemplates of SalesprojectAnalysesPhases_view.

Appearance in the client:
In the client, you can find it under Sales > Sales Dashboard > Dashlet "Opportunity phase" (by default, this Dashlet is in the lower middle part of the Sales Dashboard.

Via the button in the upper right corner of the Dashlet, you can select between the 3 chart variants: "Funnel", "Pie chart", and "Pyramid". All of them are structured by the names of the common sales project phases, e.g. "contact", "qualification", or "offer".

Configuration:


ADITO Designer - 20201007

PhaseFunnelChart - Properties

title	
devices	MOBILE | TABLET | DESKTOP
type	dynamicSingleDataChartViewTemplate
Entity	
entityField	#ENTITY
columns	COUNT
Fields	
informationField	
defaultGroupFields	#EXTENSION.Phase_filterExtention....
Others	
chartType	FUNNEL
shareParent	✓
maxDBRow	400

11.3.9. Favorite

This ViewTemplate enables you to attach "tags" to a selected dataset. This dataset will then appear in Context "Favorite", which can be accessed via the "star" button in the sidebar of the client. (Exception: Datasets exclusively tagged by *Hash*tags do not appear in Context "Favorite".)

Example:
"Favorites", a ViewTemplate of PersonPreview_view.

Appearance in the client:
Provides an area that enables you to attach "tags" to the selected dataset. The "star" toggle button adds/removes a tag titled "STANDARD". Besides, a text box is available for adding further tags with arbitrary titles (titles of existing tags appear in a dropdown menu).

Technical background:

ViewTemplate "Favorite" is exclusively to be used in PreviewViews.

Mind the wording:

- "Tag" in the broader sense means a kind of "stamp shape" that can be assigned ("stamped") to multiple records (datasets). In ADITO, this object cannot exist without at least one assignment ("stamp"). Internally, this "stamp shape" is called "record group".

- In the narrower sense, "tag" means a single "stamp" created by the "stamp shape" - i.e., a single assignment of a record group.

Technically, setting a new tag means to insert a new dataset in 2 system tables:

- ASYS_RECORDGROUP:
  - Contains datasets representing "tags" in the broader sense ("stamp shapes", see disambiguation above).
  - A "record group" can also be seen as "tag class", identified by the tag title. Whenever the client user adds a tag that has a new title, first a new "record group" is created (only *once* per title), before the tag (record group) itself is assigned (= inserted in table ASYS_RECORD, see below).
  - ID: The unique technical identificator of the record group (although the TITLE is also unique)
  - USER_ID: ID of the ADITO user who has created the record group.
  - TITLE: title of the tag
  - GROUP_TYPE: There are only 2 group types, namely
    - FAVORITE_GROUP: corresponds to a titled tag, which will make the dataset appear in Context "Favorite" under a group node having the same title
    - DEFAULT_FAVORITE_GROUP: corresponds to a "default" tag that the client user can add by clicking the "star" button of the favoriteViewTemplate. The tagged dataset will appear in Context "Favorite" under a group node having the title "STANDARD".
    - HASHTAG: corresponds to a tag whose title starts with "#". Datasets having these kind of tags will not appear in Context "Favorite", but they can be used as search criteria when performing an indexsearch.
- ASYS_RECORD:
  - Contains datasets representing "tags" in the narrower sense ("stamps", see disambiguation above).
  - Whenever, in the client, you add a tag to a record (dataset), a new dataset in ASYS_RECORD is created. In other words, a row in this table represents an assignment of a "record group" (see above) to a record.
  - ID: The technical identificator of the tag assignment.
  - OBJECT_TYPE: The text that will appear in Context "Favorite" in column OBJECT TYPE. Usually, this will be the name of the Context that holds the dataset that is tagged (e.g.

"Person") - see paragraph "Configuration" below.

  - ROW_ID: The primary key of the dataset that is tagged, e.g., the OFFERID of a dataset of Context "Offer".
  - RECORDGROUP_ID: The ID of the record group (i.e., of a dataset of table ASYS_RECORDGROUP, see above) that is used for the tagging. In other words, this is the ID corresponding to the "stamp shape" that has been used to produce the "stamp".

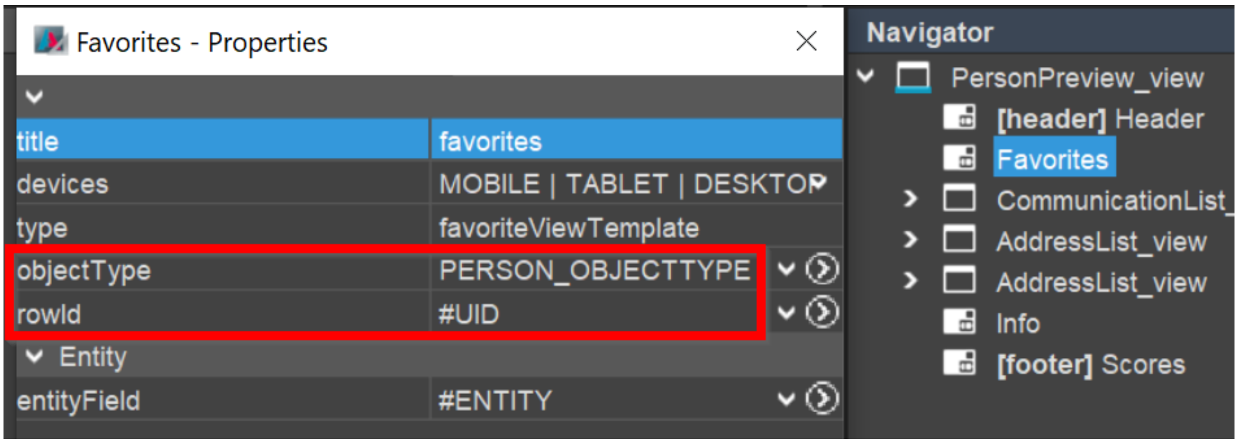
As you can see, this data structure allows the client user to use one tag title (record group) for tagging datasets of various Contexts. For example, you could tag an Activity with the title "urgent" and then use the same title for tagging datasets of Context "Knowledge". In Context "Favorite", both datasets will appear under the same grouping node

Configuration:

Make sure that the corresponding Entity’s property "useFavorites" is checked (in order to enable "private" tags). As for the ViewTemplate itself, you normally have **nothing to configure**. Simply name the favoriteViewTemplate "Favorites" and place it at the preferred position in your Context’s PreviewView. According to AID003 Design Guideline, it should be generally be placed directly under the PreviewView’s topmost ViewTemplate "Card".

That’s all - because the central properties "objectType" and "rowId" have default values that are fitting for most use cases (see below).

Exceptionally, you can configure the ViewTemplate’s properties manually, as it was done in some Contexts of earlier ADITO versions:



title	favorites
devices	MOBILE | TABLET | DESKTOP
type	favoriteViewTemplate
objectType	PERSON_OBJECTTYPE
rowId	#UID
Entity	
entityField	#ENTITY

- title: can be left empty, as this property is not used by the logic (NOTE: The title is NOT the drawer caption "TAGS", which appears in the client. Currently, this drawer caption is preset by the ADITO core and cannot be changed. It is planned to introduce an additional property "drawerCaption", with which you can optionally overwrite this default value.)

- objectType: Name of the EntityField holding the "object type" - i.e., the text that will appear in Context "Favorite" in column OBJECT TYPE. Usually, this will be the name of the Entity’s Context, e.g. "Person" (in the above example).

  Simply name the EntityField "<name of Context>_OBJECTTYPE" (e.g., PERSON_OBJECTTYPE) and configure it with a tiny valueProcess:

  Person_entity.PERSON_OBJECTTYPE.valueProcess

  ```
  result.string("Person");
  ```

  In most cases, you can simply leave the default value "#CONTEXTNAME" instead of selecting an EntityField holding the Context’s name. Internally, "#CONTEXTNAME" will automatically be replaced by the name of the Context, e.g. "Activity".

- rowId: The identificator of the record (dataset) to be tagged - e.g. "ACTIVITYID" for records of Context "Activity", or "CONTACTID" for records of the Contexts "Organisation" or "Person".
  In most cases, you can simply leave the default value "#UID" instead of the actual primary key column. Internally, "#UID" will automatically be replaced by the primary key of the table marked as "UID Table" in the DBRecordContainer’s property "linkInformation".

Some use cases require tags to be set not by the user via the favoritesViewTemplate, but via an Action or via an automatism. How this can be customized, is explained in chapter Tags.

11.3.10. Gantt

Displays an editable Gantt chart, with the data source being specific fields of a specific Entity. The Gantt chart represents a project, which consists of multiple steps. Each step has a start date and an end date and optionally a predecessor. Steps without predecessor are located at the top level of the chart, and, depending on property stepColorField, they can be shown in another color than their successors.

In the left part of the Gantt chart you see various steps, which can be in a hierarchy, shown as a tree. To the right of each step there is a bar representing the duration of the step, shown in a calendar, which is the headline of the chart.

The hierarchy of the Gantt chart depends on the settings of the properties

- predecessorIdField: This property defines the EntityField that holds the id of the parent step
- isSubstep: This boolean property defines the EntityField that holds the information ("TRUE" or "FALSE"), whether or not the step is a substep of the step defined as predecessor. If it is a substep, the corresponding bar is shown integrated inside the bar of the parent step. If it is no

substep, it is shown below the bar of the parent step.

When the Gantt is in edit mode, the client user can move the bars back and forth, in order to change the start and end date of the corresponding steps.

In principle, the Gantt ViewTemplate can get its data from various RecordContainers, but in practice, often a JDitoRecordContainer is used.

Example:
"AllCampaignsOverviewGantt", a ViewTemplate of CampaignPlanning_view.

Appearance in the client:
In the client, you can find it under Marketing > Campaign Planning

Configuration:

This is a relative simple example of a Gantt ViewTemplate. It has the following properties set:

- columns: Here you can specify arbitrary EntityFields to be shown in the left part of the Gantt chart.
- uidField: the identificator of an activity, which also is used for the parent/child mapping of the tree
- titleField: the title to be displayed in the bar
- descriptionField: the tooltip of an activity
- beginDateField: the start date of the step
- entDateField: the end date of the step
- predecessorIdField: see above

Note that this Gantt ViewTemplate has no settings for property isSubstep. Therefore, each step is located one level below its parent step (defined by predecessorField), in a tree structure. Furthermore, this Gantt ViewTemplate is not editable, i.e., you cannot move its bars.

The various EntityFields of CampaignPlanning_entity are controlled by a JDitoRecordContainer. If you study its contentProcess and its other processes, you can learn how the values of the EntityFields are calculated.

11.3.11. Generic

A ViewTemplate of type "Generic" displays one or multiple EntityFields of one single dataset, in a vertical list. You can select arbitrary EntityFields in property "fields": Open this property’s editor and add fields using the plus ("+") button. Afterwards, you can change them via the fields' combo boxes. To remove a field, select it (checkbox) and press the minus ("-") button. You can change the order of the fields by selecting them and moving them up or down with the arrow up/down buttons.

Optionally, further properties can be set:

- title: title of the ViewTemplate, as specified in the "Add" dialog. Currently, this property has no effect.
- editMode: check, if the fields should be editable
- showDrawer: check, if the fields should appear on a drawer (= caption bar on the top of the fields, with the possibility to hide/show the fields by "folding" them in/out).
- drawerCaption: caption shown in the bar on the top of the drawer
- fixedDrawer: check to prevent the fields to be hidden ("folded in")
- hideLabels: check to hide the labels to the left of each field
- informationField: optional EntityField, which will be shown in edit mode, on the top of the other fields
- hideEmptyFields: controls whether or not a line with the label (title) of an EntityField is still to be displayed, even if the EntityField has no value (= if it is "empty").

This ViewTemplate is suitable for being used as "Footer" of a HeaderFooterLayout (see chapter Layouts).

Example:
"Info", a ViewTemplate of OrganisationPreview_view.

Appearance in the client:
In the client, you can find it under Contact Management > Company. Make sure that the "Preview" button (eye icon) is active (= framed blue). If you click on a company in the table of the Filter View, you see the Generic template "Info" in the lower part of the Preview View: It shows 4 labeled fields: Language, status, type, and information.

Configuration:
"Info" has 4 fields of Organisation_entity specified in property "fields": LANGUAGE, STATUS, TYPE, and INFO. Furthermore, the "showDrawer" property flag is checked.

11.3.12. GenericMultiple

The ViewTemplate "GenericMultiple" is used to show and enter data of the "n" part in a 1:n data relation: It displays one or multiple EntityFields of n datasets, ordered horizontally (like "columns"). Via a "Plus" button, further datasets can be created, in which the values of the specified fields can be set.

It is recommended to assign GenericMultiple ViewTemplates only to Edit Views.

Example:
"MultipleEdit", a ViewTemplate of CommunicationMultiEdit_view (Context "Communication"). This ViewTemplate is referenced by OrganisationEdit_view in Context Organisation.

Appearance in the client:
In the client, you can find it under Contact Management > Company, if you add a new company dataset (via the blue "Plus" button). "MultipleEdit" is shown in the line labeled "Communication" and offers the possibility to select a medium (e.g., email) and enter this medium’s value (e.g., "info@adito.de"). By clicking on the "plus" button to the right of the label, you can add further media data - hence the name "Generic *Multiple*".

Configuration:
"MultipleEdit" has 2 fields of Communication_entity specified in property "columns": MEDIUM_ID and ADDR. That’s all.

The label of a GenericMultiple ViewTemplate can be specified in its property "title". If not set, property "title" of the Entity is used.

Step-by-step Example:

Here is another example how a GenericMultiple ViewTemplate can be used for Actions that enable to add multiple datasets.

The example task: There should be an Action that enables us to add multiple persons (Person datasets) to a Campaign. If the Action is executed, a GenericMultiple ViewTemplate is opened, in which the persons can be selected.

For solving the task, you need 2 further Contexts and Entities: One Entity for processing the data (person are added as participants) and the second for presenting the GenericMultiple ViewTemplate for the persons.

The Entity controlling the GenericMultiple ViewTemplate is configured as follows:

- EntityFields; UID, CONTACT_ID
- Consumer: Persons → Entity Person, Provider: Contacts
- RecordContainer: jDito
  - recordFieldMappings: Add the field UID
  - jDitoRecordAlias: Data_alias

    - contentProcess (to avoid errors when saving):

      ```
      if(vars.exists("$local.idvalues") && vars.get("$local.idvalues"))
      {
          result.object([vars.get("$local.idvalues")]);
      }
      ```

    - onInsert: Write any string in the result set. (This process must not be empty, otherwise an error will occur.)

- EntityField CONTACT_ID:
    - Consumer: Persons
    - displayValueProcess

      ```
      result.string(ContactUtils.getTitleByPersonId(vars.get(„$field.CONTACT_ID“)));
      ```

- View for GenericMultiple:
    - Create a View and assign a GenericMultiple ViewTemplate to it. Select EntityField CONTACT_ID for property "columns".

The Entity responsible for saving is configured as follows:

- Consumer: Participants
    - state: EDITABLE
- Parameter: CampaignId_param (includes the ID of the Campaign, from which the Action was executed)
- RecordContainer: datalessRecordContainer
    - alias: Data_alias
- Action: addToCampaign
    - onActionProcess:

      ```
      var participants = vars.get("$field.Participants.insertedRows");
      var campaignId = vars.get("$param.CampaignId_param");
      var inserts = [];
      var table = "CAMPAIGNPARTICIPANT";
      var cols = [
                  "CAMPAIGNPARTICIPANTID",
                  "CONTACT_ID",
                  "CAMPAIGN_ID",
                  "USER_NEW",
                  "DATE_NEW"
              ];

      participants.forEach(function(oneParticipant)
      {
          let contactId = oneParticipant["CONTACT_ID"];
      ```

```
    inserts.push([table, cols, null, [util.getNewUUID(), contactId, campaignId, vars.get("$sys.user"), vars.get(
"$sys.date")]]);
});

db.inserts(inserts);
```

- View for showing GenericMultiple:
  Add the GenericMultiple View, which is referenced by the Consumer "Participants", to the View via "Add reference to existing View".

Example with one single Entity:

Unlike in the above example, it is also possible to implement a GenericMultiple ViewTemplate with one single Entity. In the xRM project, you find an example in Context VisitRecommendationNewVisitplanEntry. Here, the Entity has 2 RecordContainers and a Consumer VisitRecWithNoTimes connected to Provider VisitTimes of the same entity.

11.3.13. IndexSearch

A ViewTemplate of type "IndexSearch" displays a field for entering search terms to be executed by the ADITO index search.

Example:
"IndexSearchTemplate", a ViewTemplate of IndexSearch_view (Context "IndexSearchContext").

Appearance in the client:
In the client, you can reach it by clicking on the search button in the Global Bar.

Configuration:
"IndexSearchTemplate" has its property "entityField" set to INDEXSEARCHFIELD.

11.3.14. Lookup

Enables you to integrate lookup functionality into a View, meaning the option to select a specific dataset of another Entity via its LookupView (= the View selected in property "lookupView" in the corresponding Context).

The UID of the selected dataset will then be assigned as value of the EntityField specified in property "consumerField". This EntityField must have a Consumer that is connected with a Provider of the other Entity that holds the datasets from which the user wants to select.

Example:

Salesproject > SalesprojectPhase_view > Phases
Here, a LookupView is used to show a StepperViewTemplate in the "Detail" area of the MainView.

Appearance in the client:

The appearance is defined by the LookupView of the Provider Entity. in the above example, it’s the SalesprojectPhaseStep_view, which includes a ViewTemplate of type "Stepper".

Configuration:

- "consumerField": The EntityField to be used for storing the value of the selected dataset.
- "consumerPresentationMode": Mode to be used for presenting the LookupView provided via the Consumer.
  - POPUP (default): The LookupView appears as separate (popup) window.
  - EMBEDDED: The LookupView appears embedded, i.e., it is integrated into the actual View (no need for clicking to open it).

11.3.15. Map

Displays a map, with zoom controls.

The map is provided by a datasource, usually called "map server" (or "tile server"). On this map, the following elements can be added (and, if required, be displayed):

- Markers: These are points (positions) on the map, given by geodetic coordinates (latitude und longitude, short form: "LON/LAT"). Example: Positions of companies, identified by their addresses.
- Radius around point. Example: Action "Radius Search" in PreviewView of Context "Company" (Organisation_entity). In this example, according to the entered radius, a circle around the company’s standard address is added to the map, in order to calculate the locations of all companies inside this range. By clicking the button you can show these companies in the FilterView - but the circle itself remains invisible, unless you switch to ViewTemplate "Map".
- Polylines and polygons: These are colored drawings in form of one or multiple connected lines, in order to, e.g., mark/display the boarders of a distribution area - optionally augmented by interactive functionality.

Example:
"OrganisationMap", a ViewTemplate of OrganisationFilter_view.

Appearance in the client:

In the client, you can find it under Contact Management > Company > FilterView > view selection button > "Map". It shows the locations of all companies, or of the actual filter result (if a filter is set), respectively. The maximum of locations to be shown can be configured (see below). Usually, you first define the companies you want to see in the map, by filtering them in ViewTemplate "Table", and then switch to ViewTemplate "Map".

Configuration:

Prerequisite:
The geo coordinates of the companies' standard address must be provided via Organisation_entity’s EntityFields STANDARD_LON (longitude) and STANDARD_LAT (latitude). As these reference the corresponding EntityFields "LON" and "LAT" of Address_entity, those must be given in the corresponding database table ADDRESS' columns "LON" and "LAT". Whenever you create a new ADDRESS dataset in the client, the xRM logic automatically calculates and inserts the values of the "LON" and "LAT" fields - see property "onDBInsert" of Address_entity’s RecordContainer "db". In particular, consider the code
`new LocationFinder().getGeoLocation(address)`
which makes use of functionality provided by the library Location_lib (in project folder process > libraries), on the basis of the Nominatim API.
Note that this automatism is only active, if property "nominatim.enabled" is set to true and the nominatim server is configured (see "Projects" window: preferences > PREFERENCES_PROJECT > Custom > PREFERENCES_PROJECT > nominatim.xxx). Otherwise the LON/LAT values will not be calculated automatically, and the corresponding address cannot be shown on a map.

If an address dataset has *not* been created via the client (but, e.g., imported from another system without automatic setting of LON/LAT values), you can subsequently insert all missing geo coordinates by executing the server process "Set missing address locations" in the ADITO Manager (internal name: setMissingAddressLocations_serverProcess, see project folder process > executables).
If you want to set/update the geo coordinates of **all** datasets (including overwriting all possibly existing LON/LAT values), execute the server process "Set all address locations" (internal name: updateAllAddressLocations_serverProcess).

The configuration of the map server is set via several properties, defining the source of the geo coordinates of the companies' addresses, as well as further features. All properties are well-documented via the property descriptions (at the bottom of the "Properties" window), so you can learn the functionality by studying this example.

In particular,

- property "configField" holds the EntityField providing the configuration of the map data source (named "map server" or "tile server"). In this case, this is Organisation_entity’s EntityField MAP_CONFIG, whose value is generated by its valueProcess, using functions of xRM’s library MapViewTemplate_lib (see process > libraries). You may inspect the configuration’s (JSON) format and content by simply including a logging (e.g., `logging.log("MAP_CONFIG: " +`

`res);` above the last (result) code line of the valueProcess and then watching the server output, when you zoom in and out in the map (in the client). Find more information in the following sub-chapters.

- properties "autoGeneratedMarkerLatitudeField" and "autoGeneratedMarkerLongitudeField" provide the geo coordinates of the markers (in this case: of the companies' locations) to be displayed in the map. By default, the markers are displayed by a default icon, hard-coded in the ADITO platform's logic.
- property "maxDBRow" allows you to set a limit for the number of datasets (here: company addresses) to be displayed in the map.
- If you want to customize the "pin"-type marker icon or define separate icons for every marker, you can specify an EntityField providing the icon's image data, which can have 2 formats:
  - Usually, you choose a "NEON" or a "VAADIN" icon from the list of available icons, as included in the combo box of every property named "icon". The image data is then defined by simply providing the name of this icon, as a String - e.g., "VAADIN:FACTORY". The icon will still be displayed inside a "pin".
  - Alternatively, you can provide the image data in base64 String format. This allows you to use arbitrary icon images (not only a "pin") - but note that, in this case, property autoGeneratedMarkerColorField will have no effect.
- property "autoGeneratedMarkerColorField" allows you to specify an EntityField providing the color for the marker icon (if you want another color than the default color). The color must be provided as String including an ADITO color code - simply refer to the list items, as included in the combo box of every property named "color", e.g., "priority-high-color". If this value is provided by the EntityField's valueProcess instead, you may use the corresponding JDito color constants, such as `neon.PRIORITY_HIGH_COLOR`.
- property "geoJsonFeatureCollectionField" optionally holds an EntityField to provide an array of GeoJSON FeatureCollections, in order to add additional elements to the map. Find more information in the property description (bottom of property sheet) and on https://geojson.org/. Web sites like https://geojson.io/ support you to design the GeoJSON FeatureCollection according to your requirements. Furthermore, web sites like https://github.com/isellsoap/deutschlandGeoJSON offer prepared GeoJSON FeatureCollections for, e.g., displaying borders of specific countries, states, regions, or districts.
  In our example (ViewTemplate "OrganisationMap"), this property "geoJsonFeatureCollectionField" holds Organisation_entity's EntityField MAP_FEATURE_COLLECTION, whose valueProcess reads the value from Parameter MapViewAdditionalFeatures_param. This Parameter is set, e.g., when you execute Action "Radius Search" in the PreviewView of Context "Company" (Organisation_entity). Internally, this Action is named openAroundLocation, and its valueProcess opens the View

AroundOrganisationLocation_view, belonging to Context "AroundLocation". If you select AroundLocation_entity’s Action "Open" in the Designer, you will find an example of how Parameter MapViewAdditionalFeatures_param is set - and thus, how a FeatureCollection can be created, as a GeoJSON String:

Example of creating a GeoJSON FeatureCollection (excerpt of process AroundLocation_entity.Open.onActionProcess)

```
var homeFeatureCollection = {
    "type": "FeatureCollection",
    "features": [
    {
        "type": "Feature",
        "properties": {
            "ADITO-radius": vars.get("$field.SearchRadius") * 1000,
            "ADITO-color": neon.PRIORITY_HIGH_COLOR,
            "ADITO-icon": "VAADIN:MAP_MARKER",
            "ADITO-targetContext": "Organisation",
            "ADITO-targetId": vars.get("$param.OriginUid_param"),
            "ADITO-label": ContextUtils.loadContentTitle("Organisation_entity", vars.get("$param.OriginUid_param"))
        },
        "geometry": {
            "type": "Point",
            "coordinates": [
                parseFloat(vars.get("$param.LocationLon_param"), 10),
                parseFloat(vars.get("$param.LocationLat_param"), 10)

            ]
        }
    }
    ]
};
```

This ViewTemplate requires a map server to be referenced. As the "Map" ViewTemplate is based on the "Leaflet" library, ADITO supports, in principle, any map servers that use Leaflet’s "Tile Layers" as base. In particular, ADITO xRM’s Organisation_entity (see above) provides a simplified support for *MapTiler*. Find more details in the following chapter.

11.3.15.1. MapTiler

The easiest way to configure a map data source is to use MapTiler:

- Obtain a license key for MapTiler (e.g., via https://www.maptiler.com/cloud/plans)
- Enter the license key (i.e., a String like "rf1XkCIjY4iUR4sACNjT") in the system configuration’s property geo.maptiler.apikey:
  system > default > *CONFIGURATION > Custom* >CONFIGURATION > geo.maptiler.apikey
- Save
- Re-start the ADITO server.
- Test it by opening the Company Context’s ViewTemplate "Map" (internal name:

"OrganisationMap", included in OrganisationFilter_view, see description above).

- If you want to use these functionality in other Context's, simply study the properties of ViewTemplate "OrganisationMap" and the corresponding EntityFields - and then transfer the configuration and the included logic accordingly.

11.3.15.2. General information on the required structure of map data sources

The ADITO ViewTemplate "Map" needs a data source (server) that provides basic map data to display parts of the world. The world map is structured into several parts, called "tiles". These tiles are to be provided by the server.

Find more information about tiles on https://wiki.openstreetmap.org/wiki/Tiles

The "Map" ViewTemplate is based on the "Leaflet" library (find further information and documentation on https://leafletjs.com/). Thus, this ViewTemplate is not restricted to using MapTiler (see previous chapter), but, in principle, it supports any map data source that uses Leaflet's "Tile Layers" as base.

11.3.15.2.1. Requirements

The server that provides the map layer (i.e., the map elements, such as land masses, sea, rivers, streets, etc.) must fulfill the following requirements:

- Providing raster tiles (as images) (vector tiles are not supported by Leaflet)
- Using the EPSG3857 coordinate system
- Access via an URL in "xyz"-type format (xy-coordinates of the position, plus zoom level).
- Using either no authentication at all, or authentication via the URL, by providing a license reference such as an appid, apikey, appcode, etc.

11.3.15.2.2. Property "configField"

The "Map" ViewTemplate's core property for referencing the map server is property "configField". Here, you reference an EntityField providing all information required for the ADITO web server to load map data from an (mostly: external) map server (such as MapTiler). The value of this EntityField must have JSON format. The JSON String has several parts, which are explained in the property description (in the "Properties" window, click on "configField" and read the description at the bottom of the property sheet).

Here is an example:

Example of map server configuration, as provided via property "configField"

```
{
```

```
"startingCenterPosition":{"lat":50.989791,"lon":4.772377,"autoLocate":true,"zoomLevel":5},
"boundaries":{"minZoom":1,"maxZoom":20},
"tiles":[{"title":"Streetmap",
          "url":"https://api.maptiler.com/maps/streets/256/{z}/{x}/{y}.png?key=rf1XkCIjY4iUR4sACNjT",
          "attribution":"<a href=\"https://www.maptiler.com/copyright/\" target=\"_blank\">&copy; MapTiler</a>
                         <a href=\"https://www.openstreetmap.org/copyright\" target=\"_blank\">&copy; OpenStreetMap contributors</a>"
        }]
}
```

11.3.15.2.3. URL

The core element of this JSON String is "url", e.g., `https://api.maptiler.com/maps/streets/256/{z}/{x}/{y}.png?key=rf1XkCIjY4iUR4sACNjT`

As you can see, the URL given in the JSON String can have multiple parts. The main parts are

- a reference of the map server, i.e., a "web page" - e.g., `https://api.maptiler.com/maps/streets/256/`
- placeholders for the geo coordinates (`{x}` and `{y}`) and for the zoom factor (`{z}`)
- an extension specyfing the image format, e.g. `.png`
- a suffix, attached by ?, specifying the license key, e.g., `?key=rf1XkCIjY4iUR4sACNjT`

Placeholders

You can find detailed information in the "Leaflet" documentation, available on https://leafletjs.com/ Here is an extract covering the placeholder topic:

```
{s} means one of the available subdomains (used sequentially to help
with browser parallel requests per domain limitation; subdomain
values are specified in options; a, b or c by default, can be
omitted), {z} — zoom level, {x} and {y} — tile coordinates. {r} can
be used to add "@2x" to the URL to load retina tiles.
```

("retina tiles" are high-resolution tiles)

This means, you are not limited to the placeholders given in the above example, but the following placeholders can be used:

Table 4. URL placeholders

Placeholder	Meaning
{s}	one of the available subdomains (used sequentially to help with browser parallel requests per domain limitation, one of the following values: *a*, *b* or *c*
{z}	z-axis, which is the zoom level
{x}	x-axis of the coordinate system for specifying the tile
{y}	y-axis of the coordinate system for specifying the tile
{r}	adds an option for high-DPI-tiles

Authentication

Most tile server vendors require you to authenticate via an apikey (or similar) and to customize the tiles by passing further options via the URL.

Authentication cannot be done with oAuth since the images are pure `<img>`-HTML-tags. Instead, URL parameters must be used to specify, e.g., an appid, apikey, appcode, or something similiar.

As the URL can be viewed within the browser, any logged-in user can view the map source URL, including the key.

Example

Here is a more generic example of a possible URL specification:

```
https://www.{s}mymapsupplier.org/{x}/{y}/{z}.png?apikey=MYAPIKEY&lang
=DE&style=flat
```

11.3.15.2.4. Server flexibility

It is possible to change the configuration during run-time. The next time the ViewTemplate is opened or reloaded, the new configuration will be applied.

The configuration of the map sever can be changed by modifying the JSON String provided by the

EntityField that is specified in the "Map" ViewTemplate’s property "configField" (URL, title, copyright attribution, optional elements like min and max zoom - see paragraph "URL" above).

Thus, in principle, it is possible to enable the client user to use multiple map servers in order to switch between the different types of tiles, e.g., a satellite map and a street map.

Note that the place and way of configuring the tile server may change in future ADITO versions.

11.3.16. SingleDataChart

11.3.16.1. Overview

Displays single-dimensional data in a chart, with the data source being specific fields of a specific Entity. In property “chartType”, you can select from a list of various chart types, e.g., “DONUT”.

Example:
"SingleDataChart", a ViewTemplate of CampaignCostChart_view.

Appearance in the client:
In the client, you can find it under Marketing > Campaign, in the CampaignMain_view, Tab "Overview" (CampaignOverview_view), right below the Gantt chart. It shows, as a donut, the fix costs and the variable costs.

Configuration:
"SingleDataChart" has several fields of CampaignCostChart_entity specified in different properties

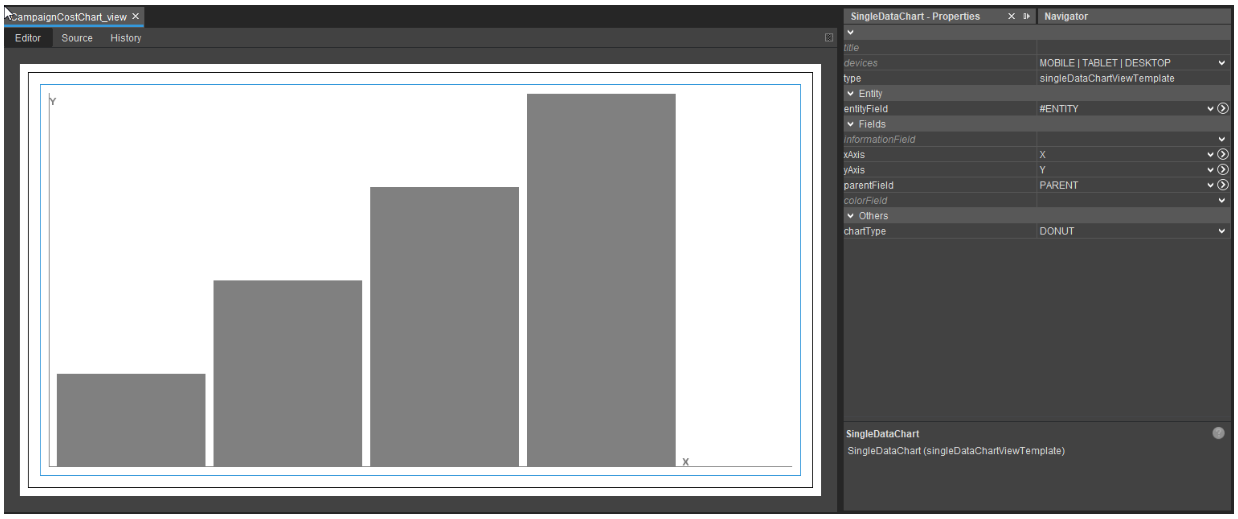
- "xAxis": X
- "yAxis": Y
- "parentField": PARENT

Besides, property "entityField" is set to #ENTITY, and property "chartType" is set to DONUT.

11.3.16.2. Advanced explanations

SingleDataChart and MultiDataChart work on the same principles, but differ in the way that they offer different chart types and that the MultiDataChart has one more property than the SingleDataChart.

Both are typically loading their data by a JDitoRecordContainer (see chapter "JDitoRecordContainer").



title	
devices	MOBILE | TABLET | DESKTOP
type	singleDataChartViewTemplate
Entity	
entityField	#ENTITY
Fields	
informationField	
xAxis	X
yAxis	Y
parentField	PARENT
colorField	
Others	
chartType	DONUT

SingleDataChart
SingleDataChart (singleDataChartViewTemplate)

Figure 32. SingleDataChart ViewTemplate

The SingleDataChart is used to display one dimensional data. This means there may only be one y axis value for any given x axis value. Typical examples are pie or funnel charts, where each segment represents one specific value.

11.3.16.2.1. Properties

- title
  The title
- devices
  Select on which device types this ViewTemplate should be available.
- type
  Predetermined by the Designer. Shows the type of the ViewTemplate.
- entityField
  The EntityField that should be used to gather the data for the chart. Usually, #Entity is used.
- informationField
  The EntityField that contains the label of the chart data
- xAxis
  The EntityField holding the value that determines the segment of the chart. For example, a slice of the pie chart. Thus, e.g., a value of 1 would determine the first slice, 2 the second slice, etc.
- yAxis
  The EntityField that holds the actual value of a segment, e.g., the number of customers of a certain type.
- parentField
  This is an EntityField that may hold an ID that is used to enable drill downs. With this you can

nest different data within a segment of the chart.
For example, if you have a pie chart showing the number of customers of a certain type, you could nest another pie chart within the segments, showing how many persons within a type have opted out of your newsletter emails.

- colorField
  Here you choose an EntityField that holds color information. The color must be one of the preset colors available through the color constants of the `neon.` JDito module. Typically, the color-holding EntityField is filled by a valueProcess. The color actually visible in the client depends on the Theme (see chapter Themes).

  Example of a valueProcess of a color-holding EntityField

  ```
  result.string(neon.USER_COLOR_1);
  ```

- chartType
  Here you select the type of chart. The SingleDataChart offers the following:
  - Donut
  - Funnel
  - Pie
  - Pyramid

11.3.16.2.2. Example

As an example, we create a pie chart of the distribution of contacts by type with drill down to gender distribution within the person contacts.

For this example we create a test Context and add one View. To the View we add a ViewTemplate of type SingleDataChart. Set its property "chartType" to "PIE". Then we create a test Entity and add to it a JDitoRecordContainer and five EntityFields. These five fields are named UID, X, Y, INFORMATION and PARENTID. Now we configure the JDitoRecordContainer, set the jDitoRecordAlias property to your data alias, and configure the recordFieldMappings property to be in the following order:

- UID.value
- X.value
- Y.value
- INFORMATION.value
- PARENTID.value

Then we set the contentProcess property as follows:

contentProcess

```
// First step: Gathering all required data
var countOrg = new SqlBuilder().selectCount()
            .from("CONTACT")
            .where("CONTACT.PERSON_ID is null")
            .cell();

var countPrivate = new SqlBuilder().selectCount()
            .from("CONTACT")
            .where("CONTACT.PERSON_ID is not null and
CONTACT.ORGANISATION_ID = '0'")
            .cell();

var countFunction = new SqlBuilder().selectCount()
            .from("CONTACT")
            .where("CONTACT.PERSON_ID is not null and
CONTACT.ORGANISATION_ID is not null")
            .cell();

// gathering the data for the drill downs
var genderDistPrivate = newSelect("GENDER, count(*)")
                            .from("PERSON")
                            .join("CONTACT", "PERSON.PERSONID =
CONTACT.PERSON_ID")
                            .where("CONTACT.ORGANISATION_ID = '0'")
                            .groupBy("GENDER")
                            .table();

var genderDistFunction = newSelect("GENDER, count(*)")
                            .from("PERSON")
                            .join("CONTACT", "PERSON.PERSONID =
CONTACT.PERSON_ID")
                            .where("CONTACT.ORGANISATION_ID is not
null and CONTACT.PERSON_ID is not null")
                            .groupBy("GENDER")
                            .table();

// Second step: Building our graph data
// Order is: UID, X, Y, INFORMATION, PARENTID
// First we create the array that will contain all datasets
var ret = [];

// Then we add our main graph data
ret.push([util.getNewUUID(), "ORG", countOrg, "Organisations",
null]);
ret.push([util.getNewUUID(), "PRIVATE", countPrivate, "
Organisations", null]);
ret.push([util.getNewUUID(), "FUNCTION", countFunction,
"Organisations", null]);

// Now we create two drill downs
for(let i = 0; i < genderDistPrivate.length; i++)
    ret.push([
        util.getNewUUID()
        , genderDistPrivate[i][0]
        , genderDistPrivate[i][1]
        , KeywordUtils.getViewValue($KeywordRegistry.personGender(),
genderDistPrivate[i][0])
        , ret[1][0] //getting the parent id for the private slice
    ]);

for(let i = 0; i < genderDistFunction.length; i++)
    ret.push([
        util.getNewUUID()
        , genderDistFunction[i][0]
        , genderDistFunction[i][1]
        , KeywordUtils.getViewValue($KeywordRegistry.personGender(),
genderDistFunction[i][0])
        , ret[2][0] //getting the parent id for the function slice
    ]);

// Finally, we return our data
result.object(ret);
```

You will notice in the code that the entries for the main graph have `null` as their parent id. That is because those do not have a parent and are the data that is presented first. When creating the datasets for our drill downs, we use the ID of the respective slice to group the drill down data below them.

❗ Make sure that potential parents are added to the result before any of their children.

After that, we are done with the Entity. Now we head back to our Context and set our new Entity to be used for this Context, and we set our only View for all the standard Views. After that's done, we open our View again and configure the properties of the ViewTemplate. Finally, we add the new Context in application > ____SYSTEM_APPLICATION_NEON, so we can open it in the client.

If we open this Context, we should get the following chart:

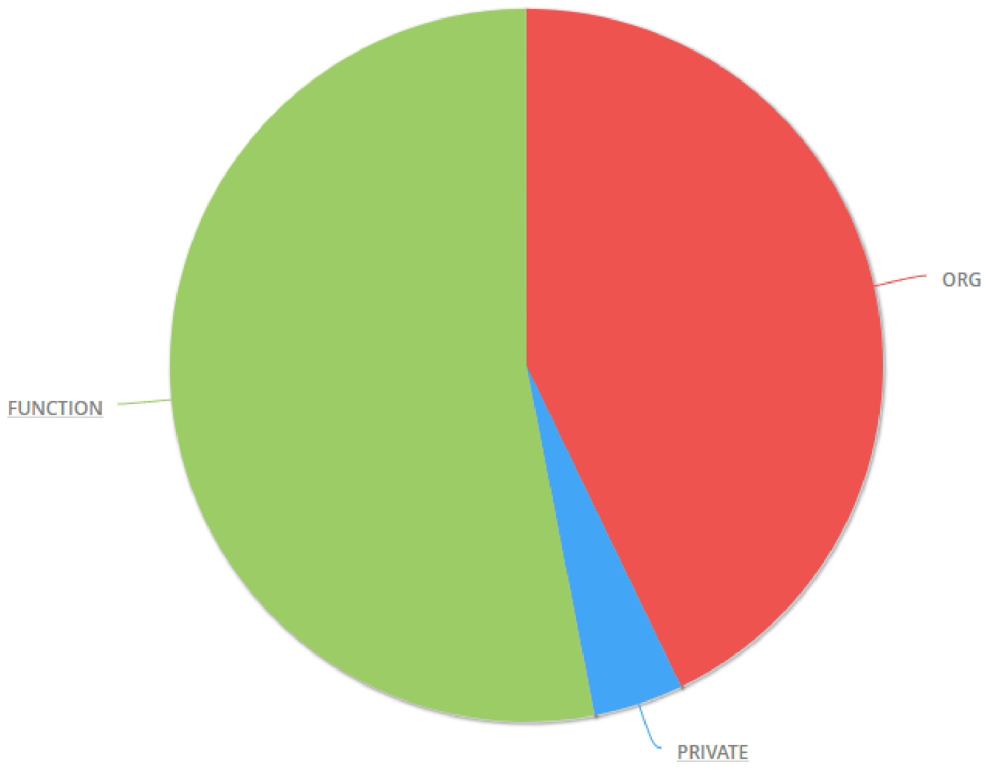


Figure 33. Main graph

After a click on the "FUNCTION" slice, we should get the gender distribution:

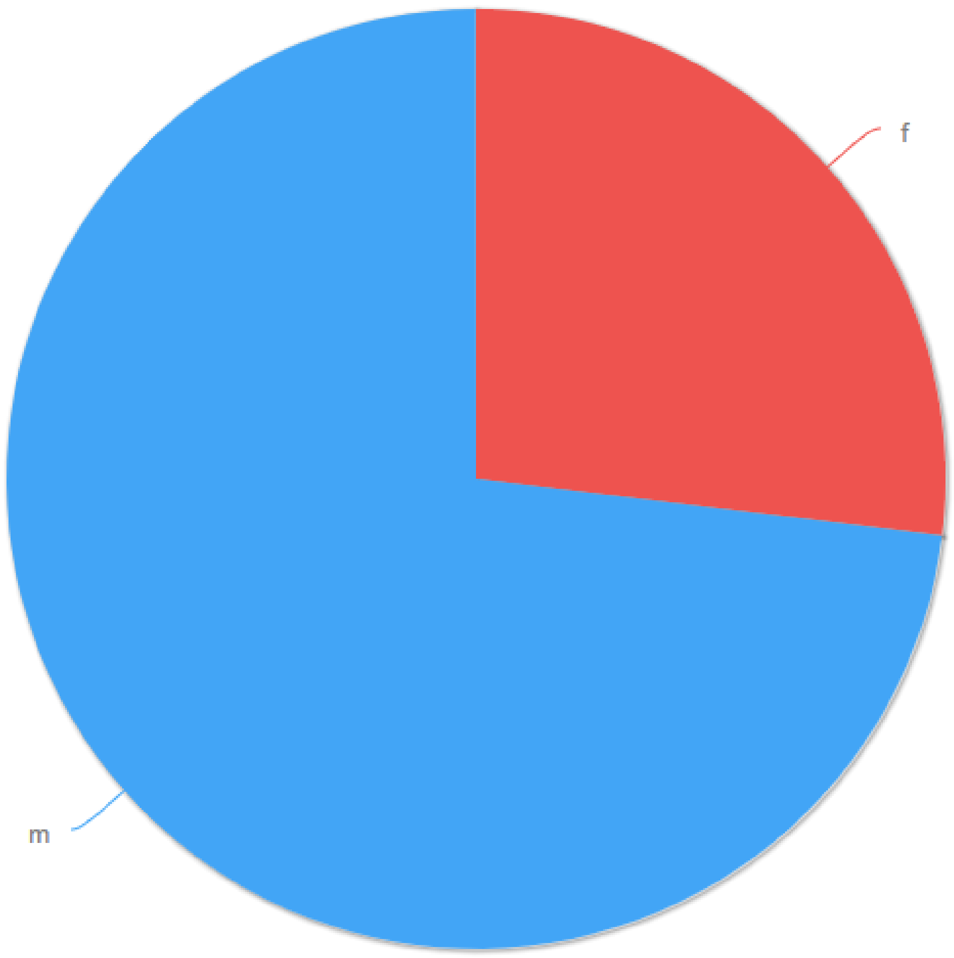


Figure 34. Drill down

You’ll notice that we have drill downs for "FUNCTION" and "PRIVATE", but not for "ORG". Now, try to do this by yourself: Add a drilldown to "ORG" that breaks down the organisation by their legal form.

11.3.17. MultiDataChart

11.3.17.1. Overview

Displays multi-dimensional data in a chart, with the data source being specific fields of a specific Entity. In property “chartType”, you can select from a list of various chart types, e.g., “COLUMN”.

Example:
"MultiDataChart", a ViewTemplate of CampaignParticipantChart_view.

Appearance in the client:
In the client, you can find it under Marketing > Campaign, in the CampaignMain_view, Tab "Overview"

(CampaignOverview_view), left below the Gantt chart. It shows, as bars, the number of current participants and maximal participants (1st dimension), separately for added participants and participants that have been contacted by telephone (2nd dimension).

Configuration:
"MultiDataChart" has several fields of CampaignParticipantChart_entity specified in different properties

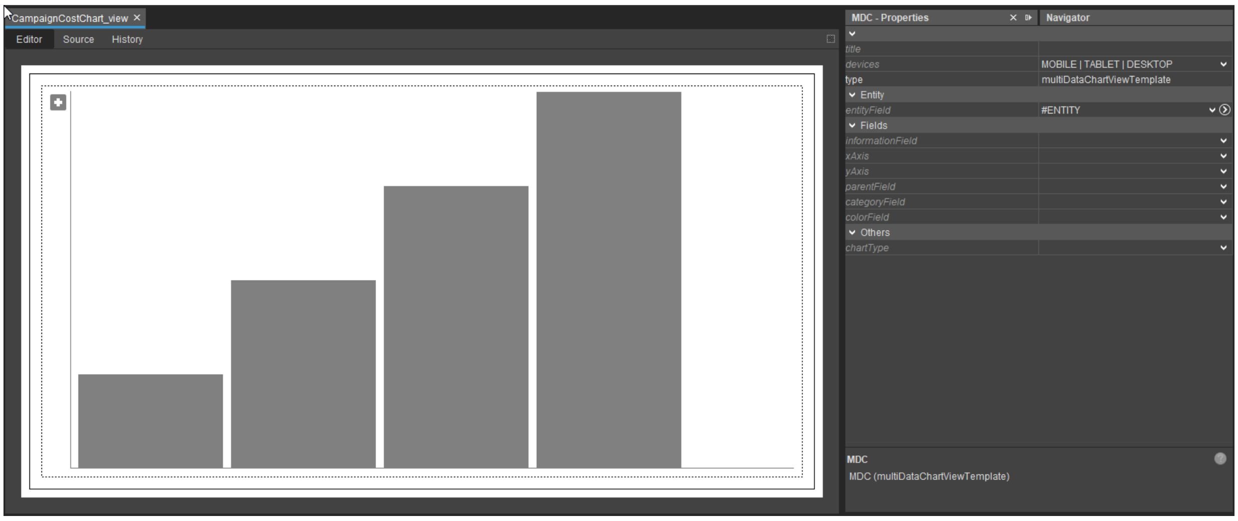
- "xAxis": X
- "yAxis": Y
- "categoryField": CATEGORY

Besides, property "entityField" is set to #ENTITY, and property "chartType" is set to BAR.

11.3.17.2. Advanced explanations

SingleDataChart and MultiDataChart work on the same principles, but differ in the way that they offer different chart types and that the MultiDataChart has one more property than the SingleDataChart.

Both are typically loading their data via a JDitoRecordContainer (see chapter "JDitoRecordContainer").



title	
devices	MOBILE | TABLET | DESKTOP
type	multiDataChartViewTemplate
Entity	
entityField	#ENTITY
Fields	
informationField	
xAxis	
yAxis	
parentField	
categoryField	
colorField	
Others	
chartType	

MDC
MDC (multiDataChartViewTemplate)

Figure 35. ViewTemplate within a View

The MultiDataCharts are used to present data on a XY graph. It is possible to display more than one Y value fo each x value, hence the name of MultiDataChart. Additionally you are able to implement drill downs.

11.3.17.2.1. Properties

- title

The title

- devices
  Select on which device types this ViewTemplate should be available.
- type
  Predetermined by the Designer. Shows the type of the ViewTemplate.
- entityField
  The EntityField which should be used to gather the data for the chart. Usually, #ENTITY is used.
- informationField
  The EntityField that contains the label of the chart data
- xAxis
  The EntityField holding the value used for the x axis. For example, the number of days it took to complete a task.
- yAxis
  The EntityField that holds the corresponding value for a value on the X axis.
- parentField
  This is an EntityField that may hold an id, which is used to created drilldowns. With this you can nest different data within a segment of the chart.
  For example, if you use a line chart to implement a burndown chart, you might implement a drilldown to present the burn down of each day.
- categoryField
  A EntityField that has to hold a category value. This allows you to define more than one value on the y axis for any given value on the x axis. E.g. if you have the number of days on the x axis, you could display multiple y values if they have a different category. This could be used to create a burndown chart, where one category would be used for the planned value and the other category would be used for the real value.
- colorField
  Here you choose an EntityField that holds color information. The color must be one of the preset colors available through the color constants of the `neon.` JDito module. Typically, the color-holding EntityField is filled by a valueProcess. The color actually visible in the client depends on the Theme (see chapter Themes).

  Example of a valueProcess of a color-holding EntityField

  ```
  result.string(neon.USER_COLOR_1);
  ```
- chartType
  Here you select the type of chart. The single data chart offers the following:

- Area
- Bar
- Column
- Spline
- Line

11.3.17.2.2. Example

As an example, we want to create a column chart showing all campaign steps and their costs (fixed costs and the costs of each step).

We start similarly to the SingleDataChart (see separate chapter further below in this manual). We build a new test Context and Entity, give it a JDitoRecordContainer, add the necessary EntityFields UID, X, Y, CATEGORY, and INFORMATION, and map them in this order within the RecordContainer. Then we create a new View for our Context containing a ViewTemplate of type MultiDataChart. Set its property "chartType" to "COLUMN". After that, we insert the contentProcess of the RecordContainer. (Other than the example of the SingleDataChart, PARENTID will not be used, because we don’t implement a drill down. Implementing a drill down works the same way as in the SingleDataChart example. For a more detailed description of the steps, please look at the SingleDataChart chapter further below in this manual.)

contentProcess

```
// We use an ID of a campaign of the xRM project's demo data.
// Instead, you could read a Parameter that is passed by a Consumer.
// But for simplicity's sake, we use a fixed id for now.
// If this ID does is no longer present, just use another ID
// of a campaign of the demo data.

var campaignId = "680de39f-7f1c-4dca-8c67-9c16c3395c3f";

// initialize required variables
var costData = [];
var ret = [];
var catKeyword = $KeywordRegistry.campaignStepCostCategory();

// gathering data from CAMPAIGNCOST and CAMPAIGNSTEP
costData = newSelect("NAME, CATEGORY, NET, SORTING")
            .from("CAMPAIGNCOST")
            .leftJoin("CAMPAIGNSTEP on CAMPAIGNSTEP.CAMPAIGNSTEPID =
CAMPAIGNCOST.CAMPAIGNSTEP_ID")
            .where("CAMPAIGNCOST.CAMPAIGN_ID", campaignId)
            .orderBy("CAMPAIGNSTEP_ID desc, SORTING")
            .table();
```

```
// calculate/get display values for our data and
// adding it to our result set
costData.forEach(function([pStepId, pCategory, pNet, pSort])
{
    // if there's no step id, the costs are the
    // fixed costs of the campaign
    if(!pStepId)
    {
        // Set "FixedCosts" as pseudo step id
        pStepId = "FixedCosts";
    }
    else
        // build the step id from the SORTING and NAME columns
otherwise
        pStepId = pSort + ". " + pStepId;

    // getting the display value for the cost categories
    // from the keyword
    pCategory = KeywordUtils.getViewValue(catKeyword, pCategory);

    // Add the data to the final result in this order:
    // UID, X, Y, CATEGORY, INFORMATION
    ret.push([util.getNewUUID(), pStepId, parseFloat(pNet),
pCategory, pCategory]);
});

// returning the data
result.object(ret);
```

This should be the result, after adding the Context to your client's Global Menu (application > _SYSTEM_APPLICATION_NEON):

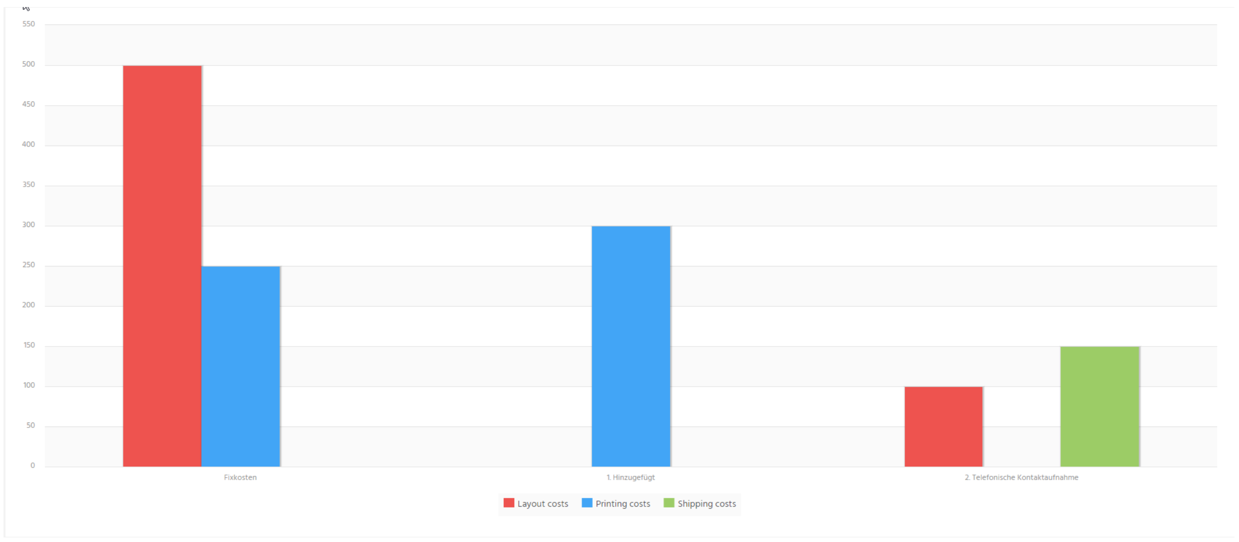


Figure 36. Resulting chart

11.3.18. MultiEditTable

A ViewTemplate of MultiEditTable type displays one or multiple datasets as editable table. Rows are datasets, columns are EntityFields. You can select arbitrary EntityFields in property "columns". Only those EntityFields defined in property "editableColumns" are displayed as editable components.

The Properties viewRendererMapping and editRendererMapping enable you to set one or multiple Renderers, in order to control the way an EntityField is displayed or edited. Please refer to chapter "Renderers" for further explanations.

Additional Information:

- Paging is not possible for this type of ViewTemplate.
- If you have changed an EntityField’s value but not saved it yet, the field is displayed with an orange frame.
- Currently, the table edit functionality is supported for EntityFields of the following contentTypes:
  - UNKNOWN
  - NONE
  - TEXT
  - NUMBER
  - TELEPHONE
  - EMAIL
  - LINK

    - PASSWORD
    - DATE
    - FILESIZE
    - BOOLEAN
    - FILTER_TREE
- EntityFields of the following contentTypes are not supported yet:
    - SIGNATURE
    - LONG_TEXT
    - HTML
    - IMAGE
    - FILE

Example:

"MultiEditTable", a ViewTemplate of ProductpriceFilter_view in Context "Productprice".

Appearance in the client and configuration:

In the Global Menu of the client, you can access the above example under Sales > Prices. As this Context’s FilterView shows a table as default, you first need to switch to the MultiEditTable, using the view selection button (upper right corner of the FilterView). This ViewTemplate displays datasets representing

- products: Selectable via a combo box, because the corresponding EntityField PRODUCT_ID has a Consumer (no Renderer required).
- prices: The corresponding EntityField PRICES can be edited via 4 additional buttons (see above), because the ViewTemplate’s property editRendererMapping specifies a Renderer of type NUMBERFIELD for EntityField PRICES.
- validity (from/to): Selectable via a date picker, because the corresponding EntityFields VALID_FROM and VALID_TO are of contentType DATE (no Renderer required).
- price list: Displayed with a background color, because the ViewTemplate’s property viewRendererMapping specifies a Renderer of type BADGE for EntityField PRICELIST. In this case, the background color is dynamically determined via the code included in the EntityField PRICELIST’s property "colorProcess". For testing purposes, you could also define a fixed color value by selecting a value for property "color" and deleting the value of property "colorProcess" (choosing "Restore Default Value" from the context menu).

11.3.19. Picture

Displays a picture, whose source data is loaded from an EntityField (property “pictureField”). Property pictureClickAction enables you to select an Action to be executed when the user clicks on the picture.

11.3.20. Report

A ViewTemplate of type "Report" displays a JasperReport. You define one EntityField as your report data source. The appearance can only be edited in your report design application.

Example:

"Report", a ViewTemplate of OfferReport_view in Context "Offer". This ViewTemplate displays the data of a selected offer.

Appearance in the client:

In the Global Menu of the client, you can access the above example under Sales > Offer. After selecting a dataset in OfferFilter_view, click on the three-dotted button in OfferPreview_view and select the Action showOffer, labeled "Show offer". This will display the report for the selected data in a separate window, using the following code (via the Action’s onActionProcess):

`OfferUtils.openOfferReport(vars.get("$field.OFFERID"));`

This method, in turn, will execute method `neon.openContextWithRecipe`:

```
var recipe = neonFilter.createEntityRecordsRecipeBuilder().uidsIncludelist([pOfferID]).toString();

neon.openContextWithRecipe("Offer", "OfferReport_view", recipe, neon.OPERATINGSTATE_VIEW, null, true);
```

Furthermore, in this example, you can optionally click on the button representing the Action dispatchOfferReport, labeled "@ Dispatch as email".

Configuration:

- "entityField": Select "#ENTITY" (or another value, according to your requirements).
- "reportData": Select the EntityField that supplies the report data (often via a valueProcess). In the above example, this is the Field "OFFER_REPORT_DATA" of Offer_entity.
- Optionally, you can specify up to 3 ActionGroups via properties "favoriteActionGroup1-3". In the above example, ActionGroup "offerReportDispatch" has been specified (which in turn includes Action "dispatchOfferReport").

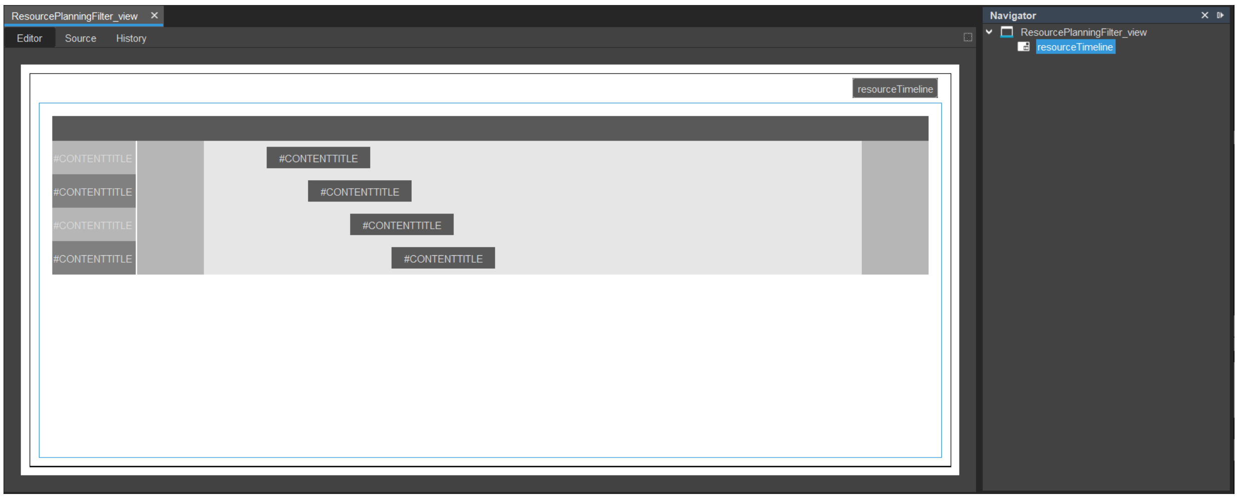
Find more information on JasperReports in the ADITO document "Reporting

Manual".

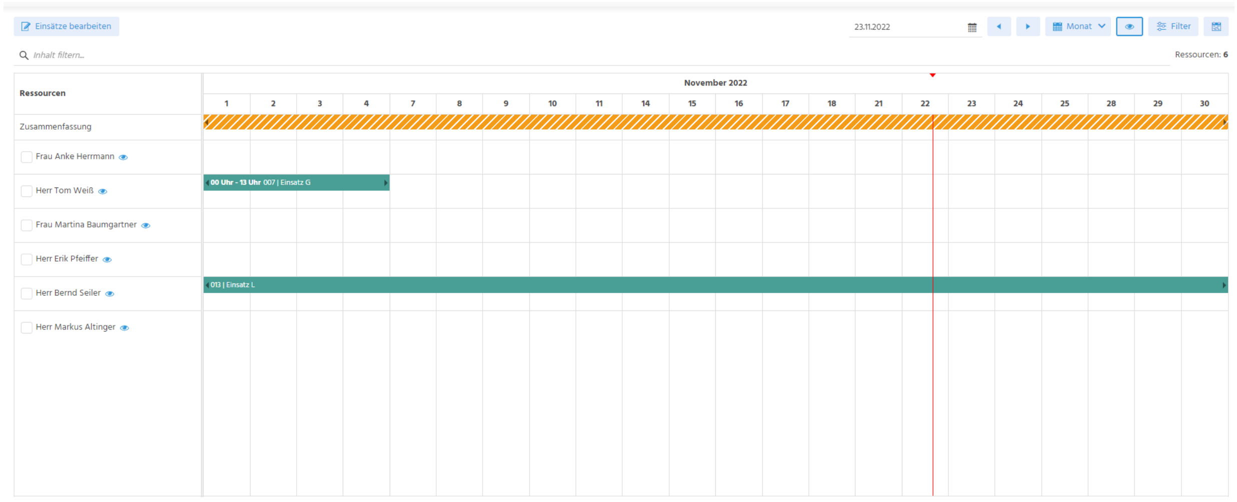
11.3.21. ResourceTimeline

Displays a calendar component that shows a tree of existing resources on the left side and visualizes scheduled operations for them on the right side. The resource timeline can be used to schedule operations. An operation in general terms is something a resource can do on a schedule like sales visits, maintenance appointments, etc.

Example:
A more advanced implementation of this ViewTemplate was done in the basic customizing model of version 2022.2.0 and above. In the Designer, it can be found in the View "ResourcePlanningFilter_view".



Appearance in the client:
It’s located under External Work → Resource planning, which is used to plan operations for existing human resources.



The ViewTemplate uses two entities to load its data. One Entity is responsible for loading the resources, the second Entity is used to load entries based on the resources and has to be connected via a Provider-Consumer pair to the Resource_entity. More detailed informations can be found in the next sections below.

11.3.21.1. Advanced explanations

The ResourceTimeline ViewTemplate is different from the other ViewTemplates in the sense that it requires two entities to get its data. Typically you'll also need at least a third one, which is used for the operations. Operations are only referenced by an UID and can be whatever you customize them to be. It can be a simple construct only using one Entity or somethig more complex like in the ADITO xRM project, where an operation can also have multiple operation tasks.

The resource planning is generally divided into three parts, which interact with each other:

- Resource: the Entity for which something is planned
- Entry: the planning of an operation for one particular resource
- Operation: the action, which is planned to be carried out

As for the relations between the three parts:

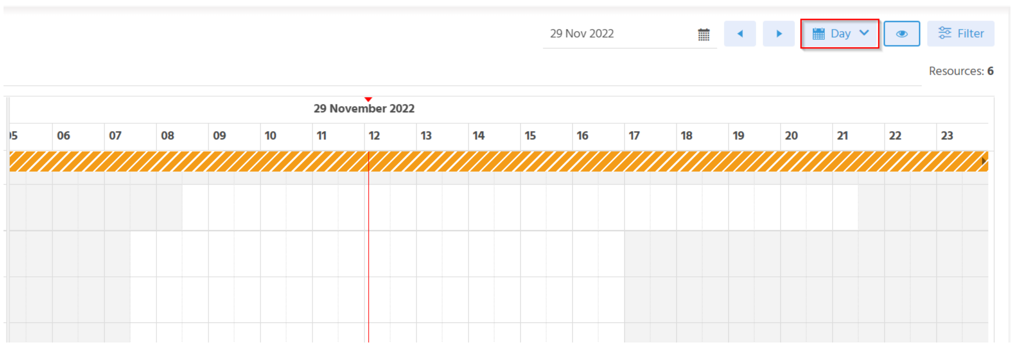
- A resource has multiple entries associated with it
- One entry is connected to one operation
- Different entries can refer to same operation

11.3.21.1.1. Important properties

The configuration offers a number of properties in order to define where to get the data. They're split up into three sections. Only the more complex ones will be mentioned below.

- Entity
  - entityField: This property is similar to other ViewTemplates and refers to the main entity of the ViewTemplate. Typically `#ENTITY` is used to refer to the Context's Entity. In this case this entity has to load the resources and is configured further in the Resources section.
- Resources
  - initalDateField: This field is used to set the shown day on opening and has to contain the date as text and has to use the ISO-8601 "yyyy.MM.dd" format. The date is parsed internally by the vietemplate. If no field is set or the set field returns NULL, the current day is used.

- parentField: As the ressource can be a tree structure, this field is used to associate child nodes to their respective parent node. If no EntityField is set or the set field is set to a NULL value, the resource is treated as a root node. This behavior is similar to the TreeTable or Tree ViewTemplates.
- businessHoursFromField: An EntityField which holds the start of the resource’s business hours as text. E.g. "08:00".
- businessHoursToField: An EntityField which holds the end of the resource’s business hours as text. E.g. "17:30".

> The values of businessHoursFromField and businessHoursToField define the timespan within which the resource is available. The timespan is shown as a white area on the timeline when using the View in day mode. Everything outside this timespan will be displayed in grey.
>
> 
>

- Entries
  - entryEntityField: This property has to hold the Consumer which should be used to load the entries. The Consumer has to be part of the Entity refered to in the `entityField` property.
  - entryResourceIdField: This property is **mandatory**! It has to hold the UID of the resource to which the entry is linked.
  - entryDatetimeStartField: This property is **mandatory**! This EntityField has to hold the start date and time of the entry as a **timestamp**. It is mandatory to configure this field to be of content type **DATE** and it has to be **filterable**.
  - entryDatetimeEndField: This property is **mandatory**! This EntityField has to hold the end date and time of the entry as a **timestamp**. It is mandatory to configure this field to be of content type **DATE** and it has to be **filterable**.
  - entrySelectedResourcesField: This field holds the resources which are selected in the component. It is provided as a JSON Array. To use it in your code the value of the field has to be processed by JSON.parse() before using it. If no resources were selected, the array

contains the id into whoses row an entry was dragged.

  - entryResourceOperationIdField: This property is **mandatory**! This field has to hold the UID of the operation to which the entry refers and is used to find the same resource operation from other resources.

11.3.21.1.2. Outlining the Entities

This section will detail the Entities needed. It will cover

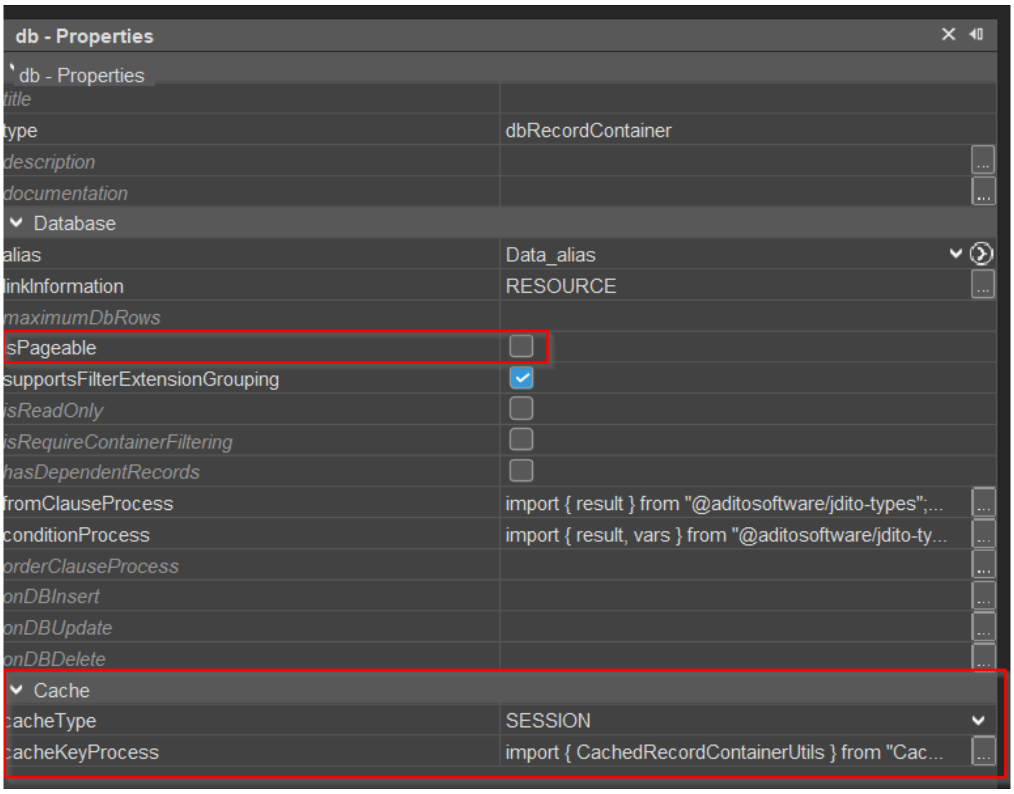
- resource Entity (in the xRM project, e.g., "Resource_entity"),
- entry Entity (in the xRM project, e.g., "ResourcePlanning_entity"),
- and operation Entity (in the xRM project, e.g., "ResourceOperation_entity").

Apart from the fields required, the Entities can be customized to your needs.

Resource Entity:

This is the main Entity of the ViewTemplate and has to provide the resources. All resource are loaded at once, so it is important to pay attention to the performance of this Entity. It also has to have a Consumer which is connected to the entry Entity. This Consumer will be used to load corresponding entries of a resource.

> The RecordContainer of this Entity has to have paging disabled, as all resources have to be loaded at once. It is also recommended to use caching to prevent loss of performance.



title	
type	dbRecordContainer
description	
documentation	
Database	
alias	Data_alias
linkInformation	RESOURCE
maximumDbRows	
isPageable	☐
supportsFilterExtensionGrouping	✓
isReadOnly	☐
isRequireContainerFiltering	☐
hasDependentRecords	☐
fromClauseProcess	import { result } from "@aditosoftware/jdito-types";...
conditionProcess	import { result, vars } from "@aditosoftware/jdito-ty...
orderClauseProcess	
onDBInsert	
onDBUpdate	
onDBDelete	
Cache	
cacheType	SESSION
cacheKeyProcess	import { CachedRecordContainerUtils } from "Cac...

Besides the fields necessary to fill the ViewTemplates properties the Entity is freely customizable and can be built to your specifications. Most of the information will be presented via the PreviewView of the Context of which this Entity is a part of and can be opened by clicking on the eye symbol besides the resource.

Creating new and editing resources have to be implemented seperately as the ViewTemplate only can show already existing records and only offers creating and editing of entries. One way is to create two separate conexts which are based on the same Entity. The first Context will have a regular Table/Treetable as FilterView where you create, edit and delete resources and the second Context has the resource timeline as FilterView within which the entries are managed. Existing resources can still be edited or deleted by using their PreviewView. This is the way this ViewTemplate is implemented in the resource planning of the basic project.

Entry Entity:

This Entity represents the planning entries which link a resource and a resource operation and has a specified start and end date. To load the correct entries this Entity has to have a Provider to which the resource Entity connects. If you don’t need more specified Providers, the `#PROVIDER` Provider can be used. The resource Entity will set a filter when requesting data. The filter is determined and set automatically and is mechanically similar to what a lookup field, i.e. an EntityField which has a Consumer chosen for providing a dropdown list, does.

If resources are selected by using the checkboxes to the left of the resource title, the IDs of these resources are set to the field specified in the entrySelectedResourcesField property. It will then contain a JSON Array of these IDs. If, for example, a new entry is created and several resources are checked, you'll have to make sure to also create an entry for every selected resource. But how this field is used and what the resulting behaviour should be, depends on your specifications. For example, if an entry should not only be created for the resource in whiches row it was created, but for all selected resources, you'll have to implement the creation of entries linked to the other resources in the afterSave process of the entry Entity or the onCreate, onUpdate and onDelete of your RecordContainer.

If no resource was selected, then the array will contain the id of the resource to whiches row an entry was dragged. You have to update the resource id of this entry, if the id you get from the array is different from the one you get from the resource id field.

In general this is the Entity where the bulk of your logic has to be implemented. E.g. creating of additional entries, updating other entries based on changes on the current entry and so on. All of these things have to be done within this Entity.

If you want to show information of the operation within the entry, the database tables of your oparation construct may be joined to the tables of the entry and set to read only mode. The read only mode is necesarry because when inserting an entry linked to an existing operation, you'd get an error because the ID of the operation already exists. If you want to include View references from your operation Context or want to select them via a lookup dropdown, the entry Entity and operation Entity should be connected via a Provider-Consumer pair.

Operation Entity:

An operation can be understood as "something a resource can do". E.g. sales visits, maintenance visits, etc. This Entity is mostly independent from the first two. Its records / data is only linked by the field set in the entryResourceOperationIdField property. Typically this Entity would have a Provider to which the entry Entity can connect for a lookup dropdown.

Apart from that, the customzing of what an operation should be, depends on your usecase and specifications. It could range from being a simple descriptor to a construct of singular plannable actions, which might have tasks or checklists associated with it and also would require more logic within the entry Entity.

11.3.21.1.3. Example: Implementing the basic functions

In the following example we will build a very basic implementation of the ViewTemplate. It will just allow to create, edit and delete resources, to create, edit and delete operations and to plan operation for a resource and move them via drag and drop.

The resources, operations and plannings will be stripped down to their bare minimum.

The resource will consist of:

- a contact, which represents the human resource for which we will plan operations
- the business hours of the resource

The operation will consist of:

- a title
- an info text field to describe the operation

The plannings will consist of:

- an operation
- the starting date
- the ending date

Creating the database tables:

Liquibase scripts and code snippets can be found in the Appendix ResourceTimeline example: Liquibase and code

We'll start by building the database tables for our three parts. Those will be called EXAMPLERESOURCE, EXAMPLEPLANNINGENTRY and EXAMPLEOPERATION.

Create a folder named resourceTimelineExample in the top level of Data_alias so the paths match with the provided scripts found in the appendix.

Within this folder create a changeset called resTimeline_creates and a changelog called changelog.
Add the changelog of this folder to the main changelog found in the top level of Data_alias.

After these liquibase scripts are in their place, rightclick on Data_alias and choose Liquibase → Update to execute the scripts. After the tables were created update the aliasdefinition, so the new tables can be used in the project.

Creating the Entities

Now we need to set up our Entities. To do that, we'll use the Blueprint `Generate Entity from`

`Aliasdefiniton` found in the context menu of the "entity" node of the project tree. Rightclick the node, then choose "New with Blueprint" → "Generate Entity from Aliasdefiniton". This will set up the Entity with a database RecordContainer and all the fields, and already connects the fields to the RecordContainer. Do that for every of our three tables. The Entities should be named ExampleResource_entity, ExamplePlanningEntry_entity and ExampleOperation_entity.

After that's done, we can go into our Entities and start by filling the valueProcesses of the fields which should automatically be filled. These fields are DATE_NEW, DATE_EDIT, USER_NEW, USER_EDIT and our ID fields EXAMPLERESOURCEID, EXAMPLEPLANNINGENTRYID and EXAMPLEOPERATIONID. The code snippets for these processes can be found in the appendix or can be taken from the according fields of other Entities. Also remove the checkmark of the mandatory property of the DATE_EDIT and USER_EDIT fields.

Necessary changes to the Entities

The Entities now need some tweaking to be complete. We'll start with the Entity, which needs the most changes: ExamplePlanningEntry_entity.

Let's start at the database RecordContainer. For the ViewTemplate to be able to select the entries based on the time window, the fields DATE_START and DATE_END have to have the `isFilterable` property checked. The same has to be done for EXAMPLERESOURCE_ID and EXAMPLEOPERATION_ID, which are also used to select the necessary planning entries. Also add the table EXAMPLEOPERATION to the linkInformation of the RecordContainer. Make sure the table is set to read only, so creating an entry won't try to also create a new opertion. As we choose our operation from a list of already created operation, the read only mode prevents errors due to already existing ids. Now that the table is linked, we can go to EXAMPLEOPERATION_ID.displayValue and select the database column "TITLE" as value of the "recordfield" property.

After that's done, we're moving on to adding a new EntityField called "SelectedResources". The ADITO core will provide a JSON array within this field, which either contains the selected resource or if no resource was selected, it contains the id of the resource this particular entry was dragged to. It will can be used to create additional entries if multiple resources were selected and is always used to implement the drag & drop functionality. This example will only implement the drag & drop functionality for simplicities sake. Creation of multiple entries can be found within the resource planning implementation of the basic project of version 2022.2.1 and above. To complete the setup of darg & drop, we have to add a valueProcess to the EXAMPLERESOURCE_ID field.

valueProcess of EXAMPLERESOURCE_ID

```
import { result, vars } from "@aditosoftware/jdito-types";

//to respond to drag and drop of the entry
if(vars.get("$field.SelectedResources")
    && JSON.parse(vars.get("$field.SelectedResources"))
    && JSON.parse(vars.get("$field.SelectedResources")).length > 0)
{
    result.string(JSON.parse(vars.get("$field.SelectedResources"))[0]);
}
```

This valueProcess will update the foreign key every time the entry is dragged to another resource.

Now we also have to change the fields DATE_START and DATE_END as thsoe have to be filled by the user. So we have to set the properties of the "Formatting" group. We'll set "resolution" to MINUTE and both format properties to the "dd.MM.yyyy HH:mm" pattern.

As the last changes we'll add the possibility to choose our operations. So we add a new Consumer to the Entity and call it ExampleOperations. The Consumer has to point to ExampleOperation_entity and use its #PROVIDER Provider. After the Consumer is configured, we go to the EntityField EXAMPLEOPERATION_ID and set this Consumer as value of its "consumer" property.

The next Entity, we're going to expand will be ExampleResource_entity. Here we'll start at the Entity itself and set its contentTitleProcess:

```
import {vars, result} from "@aditosoftware/jdito-types"

result.string( vars.get("$field.CONTACT_ID.displayValue"))
```

This will take the displayValue of the selected contact as its title.

After this change, we'll move on to the database RecordContainer. As all resources have to be loaded at once, we have to disable the "isPageable" property. For this simple example we ignore grouping and caching. The last thing to do within the RecordContatiner, is to configure CONTACT_ID.displayValue, so it shows the name of the persons we select. To achieve this, we need to add a process to the "expression" property:

```
import { result } from "@aditosoftware/jdito-types";
import {newSelect, SqlMaskingUtils} from "SqlBuilder_lib";

result.string(
    newSelect([new SqlMaskingUtils().concatWithSeparator(["FIRSTNAME","LASTNAME"]," ",true)])
    .from("PERSON")
    .join("CONTACT", "PERSON.PERSONID = CONTACT.PERSON_ID")
    .where("CONTACT.CONTACTID = EXAMPLERESOURCE.CONTACT_ID")
    .toString()
);
```

To make the resources filterable by their names, we also check the "isFilterable" property of CONTACT_ID.displayValue.

Following these changes, we’ll add two Consumers to the Entity. The first Consumer will be called ExampleEntries and has to point to ExamplePlanningEntry_entity and use its #PROVIDER Provider. This Consumer will be used by the ResourceTimeline ViewTemplate to load the entries for each resource based on the resource id and the selected time window. The second Consumer will be named "Persons" and has to point to Person_entity and use the "Contacts" Provider. This will allow us to select the contact of the resource via a lookup table. After the Consumers being set up, go to the CONTACT_ID EntityField and set the "Persons" Consumer as value of its "consumer" property.

Finally, we need to change ExampleOperation_entity. Here we’ll also start at the contentTitleProcess of the Entity and add the following code:

```
import { result, vars } from "@aditosoftware/jdito-types";

result.string(vars.get("$field.TITLE"));
```

This uses the TITLE of the Operation, so we can see its name when using the lookup within the entry. As this is done, we move on to the RecordContainer. Here we go to TITLE.value and check its "isFilterable" and "isLookupFilter" properties.

Creating Contexts and Views

Our logic and therefore Entities are now complete, and we can move on to create the frontend for them. We will need **four** Contexts for that. Two of those Contexts will reference ExampleResource_enity and be different frontends with different purposes. As we need to create, edit, and delete resources, we will do that within its own context, so it’s separated from the ResourceTimeline, which will handle the creation, editing and deletion of our entries.

We’ll add our Contexts by using the "Create Contexts with Default Views" Blueprint. It’s found in the context menu of the "context" node of the project tree. Right-click on the node and choose "New with Blueprint" → "Create Contexts with Default Views".

Our four Contexts and their Views will be:

- ExampleResource
  Set ExampleResource_entity as the used Entity.
  Check the following Views:
  - Edit_view
  - Filter_view
  - Preview_view
- ExamplePlanning

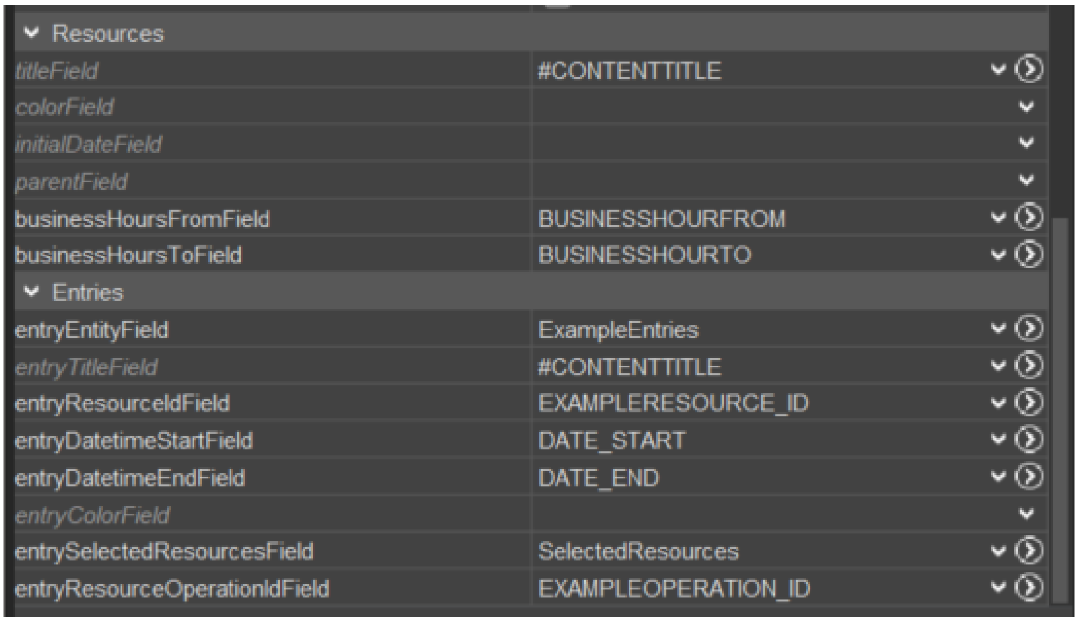
  Set ExampleResource_entity as the used Entity.
  Check the following Views:
    - Filter_view
    - Preview_view
- ExampleEntry
  Set ExamplePlanningEntry_entity as used Entity.
  Check the following Views:
    - Edit_view
    - Preview_view
- ExampleOperation
  Set Exampleoperation_entity as used Entity.
  Check the following Views:
    - Edit_view
    - Filter_view
    - Preview_view

This is the minimum of Views that you need to implement. If your specific project includes more complex constructs, the resources, entries, and operations might also require a Main_view or a Lookup_view.

After the groundwork is done, we can now fill each View. Choose suitable names for the ViewTemplates at your own will.

- ExampleResource
    - Edit_view
      Add a Generic ViewTemplate and add the CONTACT_ID, BUSINESSHOURFROM, and BUSINESSHOURTO fields. Make sure the "editMode" property of the ViewTemplate is checked. As this is all we need for this example, you can also set the "size" property of the View to "SMALL", so it’s opened besides the table we’ll use in the Filter_view.
    - Filter_view
      Add a Table ViewTemplate and add CONTACT_ID, BUSINESSHOURFROM, BUSINESSHOURTO to its column property.
    - Preview_view
      Repeat the steps from creating the Edit_view, but do **not** check the "editMode" property.
- ExamplePlanning

- Filter_view
  Here we add the ResourceTimeline ViewTemplate. To configure it, we need to fill the groups "Resources" and "Entries" as shown in the screenshot below:

Resources	
titleField	#CONTENTTITLE
colorField	
initialDateField	
parentField	
businessHoursFromField	BUSINESSHOURFROM
businessHoursToField	BUSINESSHOURTO
Entries	
entryEntityField	ExampleEntries
entryTitleField	#CONTENTTITLE
entryResourceIdField	EXAMPLERESOURCE_ID
entryDatetimeStartField	DATE_START
entryDatetimeEndField	DATE_END
entryColorField	
entrySelectedResourcesField	SelectedResources
entryResourceOperationIdField	EXAMPLEOPERATION_ID

  Within "Resources" we just need the mandatory fields "titleField", "businessHoursFromField" and "businessHoursToField", as those are necessary for displaying the resources.

  Within the Entries group we need to set the Consumer "ExampleEntries" as value of "entryEntityField". This is the connection between our resources and our entries. The follwing fields are taken from ExamplePlanningEntry_entity to fill the remaining proeprties.

- Preview_view
  Create it the same way the Preview_view of ExampleResources was created as they're basically the same.

- ExampleEntry
  - Edit_view
    Add a Generic ViewTemplate and add the EXAMPLEOPERATION_ID, DATE_START and DATE_END fields to it. Check the "editMode" property of the ViewTemplate.
  - Preview_view
    Add a Generic ViewTemplate and add the EXAMPLEOPERATION_ID, DATE_START and DATE_END fields to it. Do **not** check the "editMode" property with this one.
- ExampleOperation

- Edit_view
  Add a Generic ViewTemplate and add the TITLE and INFO fields. Check the "editMode" property of the ViewTemplate.
- Filter_view
  Add a Table ViewTemplate and add the TITLE and INFO fields to its columns property.
- Preview_view
  Add a Generic ViewTemplate and add the TITLE and INFO fields. Do **not** check the "editMode" property of this ViewTemplate.

Testing the example implementation

Now we’re done implementing our example, we have to add the ExampleResource, ExamplePlanning, and ExampleOperation Contexts to the menu by adding them to a new group within the " _SYSTEM_APPLICATION_NEON" datamodel found in the "application" node of the project tree. For simplicity’s sake also add the "INTERNAL_EVERYONE" role to your new menu group. ExampleEntry doesn’t have to be in the menu unless you want to also implement a Filter_view for it, so you can have a list of existing entries. But this is optional in this example.

After that’s done, we deploy everything and check it out within the ADITO webclient. Create some resources within ExampleResource, then create some operations within ExampleOperation, and at last create entries within ExamplePlanning and test the drag & drop of entries between different resources.

For a more comprehensive implementation of the ResourceTimeline ViewTemplate, you can take a look at the source code of the resource planning included in the ADITO xRM project.

11.3.21.2. Specific color constants

There are specific color constants to color the entries of the ResourceTimeline. These colors are set by the ADITO core, but can be overridden when using a custom Theme (see chapter Themes).

Active entry colors:

- neon.RESOURCETIMELINE_ACTIVE_COLOR_1
- neon.RESOURCETIMELINE_ACTIVE_COLOR_2
- neon.RESOURCETIMELINE_ACTIVE_COLOR_3
- neon.RESOURCETIMELINE_ACTIVE_COLOR_4
- neon.RESOURCETIMELINE_ACTIVE_COLOR_5
- neon.RESOURCETIMELINE_ACTIVE_COLOR_6
- neon.RESOURCETIMELINE_ACTIVE_COLOR_7

- neon.RESOURCETIMELINE_ACTIVE_COLOR_8
- neon.RESOURCETIMELINE_ACTIVE_COLOR_9
- neon.RESOURCETIMELINE_ACTIVE_COLOR_10
- neon.RESOURCETIMELINE_ACTIVE_COLOR_11

Passive entry colors:

- neon.RESOURCETIMELINE_PASSIVE_COLOR_1
- neon.RESOURCETIMELINE_PASSIVE_COLOR_2
- neon.RESOURCETIMELINE_PASSIVE_COLOR_3
- neon.RESOURCETIMELINE_PASSIVE_COLOR_4
- neon.RESOURCETIMELINE_PASSIVE_COLOR_5
- neon.RESOURCETIMELINE_PASSIVE_COLOR_6
- neon.RESOURCETIMELINE_PASSIVE_COLOR_7
- neon.RESOURCETIMELINE_PASSIVE_COLOR_8
- neon.RESOURCETIMELINE_PASSIVE_COLOR_9
- neon.RESOURCETIMELINE_PASSIVE_COLOR_10
- neon.RESOURCETIMELINE_PASSIVE_COLOR_11

11.3.22. ScoreCard

Displays arbitrary fields of an Entity, each on a labelled card. The label is specified in the “title”/”titleProcess” property of the corresponding EntityField.

Example:

"OrganisationInformation", a ViewTemplate of OrganisationPreview_view in Context "Organisation". This ViewTemplate displays additional information about a company.

Appearance in the client:

In the client, you can find this ViewTemplate, if you open the Global Menu and then select "Company" in menu group "Contact Management". Then select any company: The PreviewView will open, and you see the ScoreCardViewTemplate as footer.

Configuration:

- fields: 3 EntityFields of Organisation_entity are referenced: 2 calculated EntityFields

("TurnoverPercentDiff" and "LastActivity") and 1 database-related EntityField ("CLASSIFICATIONVALUE")

- fieldActions: In this example empty. Here, you can enter Actions that will be executed, if you click on an EntityField of the ScoreCardViewTemplate: The first Action is mapped to the first EntityField, the second Action to the second EntityField etc.

> You can find a further implementation of a ScoreCardViewTemplate, regarding the carpool example, in chapter Example: Availability.

11.3.23. Signature

Enables the client user to write a signature with a pointing device (e.g., a mouse) and assign it to an EntityField (property “imageField”).

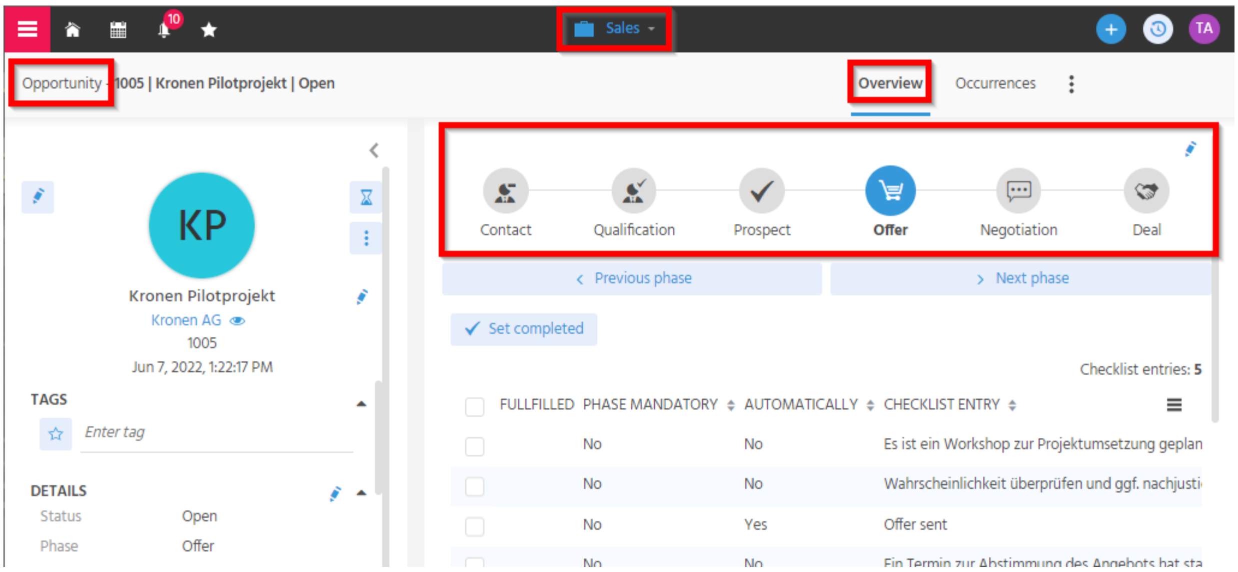
11.3.24. Stepper

Displays steps of a process, represented by titled circles, ordered in a horizonal line. Each step has an icon (property “iconField”), a title (“titleField”), and a state (“stateField”). The corresponding EntityFields ICON, STATE, and TITLE as well as an identifier (UID) are controlled by a jDitoRecordContainer.

Example:

"Phases", a StepperViewTemplate assigned to SalesprojectPhaseStep_view.

Appearance in the client:
In the client, you will find this ViewTemplate in the MainView of Context "Opportunity" (Menu Group "Sales"):



FULLFILLED	PHASE MANDATORY	AUTOMATICALLY	CHECKLIST ENTRY
☐	No	No	Es ist ein Workshop zur Projektumsetzung geplan
☐	No	No	Wahrscheinlichkeit überprüfen und ggf. nachjusti
☐	No	Yes	Offer sent
☐	No	No	Ein Termin zur Abstimmung des Angebots hat sta

Status	Open
Phase	Offer

Configuration:

Check the contentProcess of SalesprojectPhase_entity’s RecordContainer "jdito" in order to learn how the EntityFields' values are provided. Via Provider "Phases" and Parameters CurrentPhase_param, DisabledPhases_param, and SalesprojectUid_param dependencies are established from Salesproject_entity (Consumer SalesprojectPhaseStepper) and SalesprojectMilestone_entity (Consumer SalesProjectPhases). SalesprojectPhaseStep_view with its StepperViewTemplate is used as LookupView of context SalesprojectPhase. To make the StepperViewTemplate appear in the SalesprojectMain_view, a LookupViewTemplate also named "Phases" is used (see SalesprojectMain_view > SalesprojectOverview_view > SalesprojectPhase_view > Phases).

Step-by-step example:

As the above example in the Opportunity (Salesproject) Context is quite complex; another, plainer example should be explained step-by-step: Our example task is to show a contact person’s work experience in an additional tab of PersonMain_view. The steps should simply be "Beginner"/"Advanced"/"Pro". This task should be implemented by using a StepperViewTemplate without further buttons etc.

The following step-by-step example

- uses a Keyword for the steps;
- the icon will be loaded dynamically via a Keyword Attribute.
- Editing is done directly via the pencil icon. (There are no Actions to step forth/back, as there are for the Salesproject Phases, see previous example.)
- Integrating the StepperViewTemplate in the "Detail" area of a MainView requires the usage of a LookupViewTemplate (see chapter Lookup).
- Whether or not the Stepper is editable, depends on the Consumer’s state and the EntityField EXPERIENCELEVEL’s state (if its state was "AUTO", no pencil would appear).

Now, let’s solve our task step-by-step (the naming of the various new models has no technical meaning, but should be consistent, of course):

At first, we enter a new KeywordCategory (ExperienceLevel), 3 KeywordEntries (BEGINNER/Beginner, ADVANCED/Advanced, PRO/Pro), 1 KeywordAttribute (icon/ExperienceLevel), as well as 3 corresponding KeywordAttributeRelations, which will be used for assigning icons VAADIN:MINUS, VAADIN:PLUS_MINUS, and VAADIN_PLUS to the KeywordEntries. Instead of doing this manually, you better simply include the following Liquibase changelog and execute it by a Liquibase update:

```
<?xml version="1.1" encoding="UTF-8" standalone="no"?>
<databaseChangeLog xmlns="http://www.liquibase.org/xml/ns/dbchangelog" xmlns:ext="http://www.liquibase.org/xml/ns/dbchangelog-ext"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog-ext
```

```xml
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-ext.xsd http://www.liquibase.org/xml/ns/dbchangelog
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.6.xsd">
    <changeSet author="j.smith" id="5368c690-e7ce-4a0c-bb5d-bccae3aedb68">
        <!--KeywordCategory-->
        <insert tableName="AB_KEYWORD_CATEGORY">
            <column name="AB_KEYWORD_CATEGORYID" value="b1462948-f552-44f8-8408-21b7592f0902"/>
            <column name="NAME" value="ExperienceLevel"/>
            <column name="SORTINGBY" valueNumeric="0"/>
            <column name="SORTINGDIRECTION" value="ASC"/>
        </insert>
        <!--KeywordEntry-->
        <insert tableName="AB_KEYWORD_ENTRY">
            <column name="AB_KEYWORD_ENTRYID" value="7762f1ff-4e40-4022-abbe-8b73ca2abc01"/>
            <column name="KEYID" value="BEGINNER"/>
            <column name="TITLE" value="Beginner"/>
            <column name="CONTAINER" value="ExperienceLevel"/>
            <column name="AB_KEYWORD_CATEGORY_ID" value="b1462948-f552-44f8-8408-21b7592f0902"/>
            <column name="SORTING" valueNumeric="1"/>
            <column name="ISACTIVE" valueNumeric="1"/>
            <column name="ISESSENTIAL" valueNumeric="1"/>
        </insert>
        <insert tableName="AB_KEYWORD_ENTRY">
            <column name="AB_KEYWORD_ENTRYID" value="459f06bf-9623-4e85-b290-205290d6af8f"/>
            <column name="KEYID" value="ADVANCED"/>
            <column name="TITLE" value="Advanced"/>
            <column name="CONTAINER" value="ExperienceLevel"/>
            <column name="AB_KEYWORD_CATEGORY_ID" value="b1462948-f552-44f8-8408-21b7592f0902"/>
            <column name="SORTING" valueNumeric="2"/>
            <column name="ISACTIVE" valueNumeric="1"/>
            <column name="ISESSENTIAL" valueNumeric="1"/>
        </insert>
        <insert tableName="AB_KEYWORD_ENTRY">
            <column name="AB_KEYWORD_ENTRYID" value="4ba1f2a5-3cec-435f-9ac5-c0c8a13f1842"/>
            <column name="KEYID" value="PRO"/>
            <column name="TITLE" value="Pro"/>
            <column name="CONTAINER" value="ExperienceLevel"/>
            <column name="AB_KEYWORD_CATEGORY_ID" value="b1462948-f552-44f8-8408-21b7592f0902"/>
            <column name="SORTING" valueNumeric="3"/>
            <column name="ISACTIVE" valueNumeric="1"/>
            <column name="ISESSENTIAL" valueNumeric="1"/>
        </insert>

        <!--KeywordAttribute and KeywordAttributeRelations-->
        <!--icons-->
            <insert tableName="AB_KEYWORD_ATTRIBUTE">
                <column name="AB_KEYWORD_ATTRIBUTEID" value="fbb95471-0cd8-434c-b640-9fc919697fb8"/>
                <column name="AB_KEYWORD_CATEGORY_ID" value="b1462948-f552-44f8-8408-21b7592f0902"/>
                <column name="NAME" value="icon"/>
                <column name="CONTAINER" value="ExperienceLevel"/>
                <column name="KIND" value="CHAR_VALUE"/>
            </insert>

            <insert tableName="AB_KEYWORD_ATTRIBUTERELATION">
                <column name="AB_KEYWORD_ATTRIBUTERELATIONID" value="88bb3905-2013-472b-9afa-0bfd1f452951"/>
                <column name="AB_KEYWORD_ENTRY_ID" value="7762f1ff-4e40-4022-abbe-8b73ca2abc01"/>
                <column name="AB_KEYWORD_ATTRIBUTE_ID" value="fbb95471-0cd8-434c-b640-9fc919697fb8"/>
                <column name="CHAR_VALUE" value="VAADIN:MINUS"/>
            </insert>

            <insert tableName="AB_KEYWORD_ATTRIBUTERELATION">
                <column name="AB_KEYWORD_ATTRIBUTERELATIONID" value="d487ac11-b85b-4489-89e1-1e93353ad9e9"/>
                <column name="AB_KEYWORD_ENTRY_ID" value="459f06bf-9623-4e85-b290-205290d6af8f"/>
                <column name="AB_KEYWORD_ATTRIBUTE_ID" value="fbb95471-0cd8-434c-b640-9fc919697fb8"/>
                <column name="CHAR_VALUE" value="VAADIN:PLUS_MINUS"/>
            </insert>

            <insert tableName="AB_KEYWORD_ATTRIBUTERELATION">
                <column name="AB_KEYWORD_ATTRIBUTERELATIONID" value="d109b169-f14a-4383-af43-f30c6d357971"/>
                <column name="AB_KEYWORD_ENTRY_ID" value="4ba1f2a5-3cec-435f-9ac5-c0c8a13f1842"/>
                <column name="AB_KEYWORD_ATTRIBUTE_ID" value="fbb95471-0cd8-434c-b640-9fc919697fb8"/>
                <column name="CHAR_VALUE" value="VAADIN:PLUS"/>
            </insert>
    </changeSet>
</databaseChangeLog>
```

As usual in the xRM project, we will refer to the KeywordCategory and the KeywordEntries via functions. To enable this, we add the following code lines in library KeywordRegistry_basic:

```
$KeywordRegistry.ExperienceLevel = function(){return "ExperienceLevel";};
$KeywordRegistry.ExperienceLevel$beginner = function(){return "BEGINNER";};
$KeywordRegistry.ExperienceLevel$advanced = function(){return "ADVANCED";};
$KeywordRegistry.ExperienceLevel$pro = function(){return "PRO";};
```

The next steps are:

- In the database Data_alias, create a new column named EXPERIENCELEVEL for table PERSON and set the value of EXPERIENCELEVEL for all existing PERSON datasets to "BEGINNER". Instead of doing this manually, you better simply include the following Liquibase changelog and execute it by a Liquibase update:

  ```xml
  <?xml version="1.1" encoding="UTF-8" standalone="no"?>
  <databaseChangeLog xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
  xmlns:ext="http://www.liquibase.org/xml/ns/dbchangelog-ext" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog-ext http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-
  ext.xsd http://www.liquibase.org/xml/ns/dbchangelog http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.6.xsd">
      <changeSet author="j.smith" id="dcf5f410-c528-4e82-977e-91337cd4c806">
          <addColumn tableName="PERSON">
              <column name="EXPERIENCELEVEL" type="VARCHAR(36)"/>
          </addColumn>

          <update tableName="PERSON">
              <column name="EXPERIENCELEVEL" value="BEGINNER"/>
          </update>
      </changeSet>
  </databaseChangeLog>
  ```

- Update the Alias Definition, so we can refer to the new column in our project.
- Navigate to Person_entity and add an EntityField named EXPERIENCELEVEL (title = Experience level; state = EDITABLE; mandatory = true; contentType = TEXT)
- Open Person_entity’s RecordContainer (db) and connect the new EntityField with the corresponding database column: Navigate to RecordFieldMapping EXPERIENCELEVEL.value and set its property "recordfield" to PERSON.EXPERIENCELEVEL.
- EXPERIENCELEVEL.displayValue: Set the following code for property "expression" (then, the KeywordEntry’s TITLE, e.g. "Beginner", is shown instead of its KEYID, e.g. "BEGINNER")

  ```
  var sql = KeywordUtils.getResolvedTitleSqlPart($KeywordRegistry.ExperienceLevel(), "PERSON.EXPERIENCELEVEL");
  result.string(sql);
  ```

- Enter the following code in the valueProcess of EntityField EXPERIENCELEVEL, in order to preset it with BEGINNER, whenever a new Person dataset is entered:

  ```
  if(vars.get("$sys.recordstate") == neon.OPERATINGSTATE_NEW && vars.get("$this.value") == null)
  {
      result.string($KeywordRegistry.ExperienceLevel$beginner());
  }
  ```

- Enter the following code in the displayValueProcess of EntityField EXPERIENCELEVEL (then, when entering a new Person dataset, the KeywordEntry's TITLE, e.g. "Beginner", is shown instead of its KEYID, e.g. "BEGINNER"):

```
var res = KeywordUtils.getViewValue($KeywordRegistry.ExperienceLevel(), vars.get("$field.EXPERIENCELEVEL"));
result.string(res);
```

- In Context Person, create a new View named PersonExperienceLevel_view (title: Experience level; layout: BoxLayout)
- Assign a LookupViewTemplate to the new View, name it simply "Lookup", and set its properties as follows
  - consumerField: EXPERIENCELEVEL
  - consumerPresentationMode: EMBEDDED
- Open PersonMain_view and set a View reference to PersonExperienceLevel_view
- Create a new Context named PersonExperienceLevel and a new Entity named PersonExperienceLevel_entity. This new Entity will control the Stepper. Configure the Entity as follows:
  - title: Person experience level
  - contentTitleProcess:

    ```
    result.string(vars.get("$field.TITLE"));
    ```

  - EntityFields: ICON, STATE, TITLE, and UID (leave all properties in default state).
  - Parameters: CurrentLevel_param and DisabledLevels_param (set property "expose" to true for both of them).
  - RecordContainer: jDitoRecordContainer, name it jDito and set its properties as follows:
    - jDitoRecordAlias: Data_alias
    - recordFieldMappings: UID.value, STATE.value, TITLE.value, ICON.value (the order is important, as it must be consistent to the contentProcess)
    - contentProcess: Enter the following code:

      ```
      var res = [];
      var ids = vars.get("$local.idvalues");
      var disabledLevels = JSON.parse(vars.get("$param.DisabledLevels_param")) || [];
      var steps = KeywordUtils.getEntryArray($KeywordRegistry.ExperienceLevel(), null, true);
      var selected = vars.get("$param.CurrentLevel_param");

      // filter only for steps reqistered by the system
      if (ids)
      {
      ```

```
    steps = steps.filter(function(pStep)
    {
        for (let i = 0; i < ids.length; i++)
        {
            if (ids[i] == pStep[0])
            {
                return true;
            }
        }
        return false;
    })
}

steps.forEach(function([stepId, title])
{
    var stepState = "DISABLED";
    if (stepId === selected)
    {
        stepState = "ACTIVE";
    }
    else if (!disabledLevels.includes(stepId))
    {
        stepState = "EDITABLE";
    }

    var resStep = [stepId, stepState, title, _getIcon(stepId)];
    res.push(resStep);
});

result.object(res);

function _getIcon(pLevel)
{
    var iconAttr = new KeywordAttribute($KeywordRegistry.ExperienceLevel(), "icon", "VAADIN:CIRCLE-THIN");
    return iconAttr.getValue(pLevel);
}
```

- In Context PersonExperienceLevel, create a new View named PersonExperienceLevelStepper_view (layout: BoxLayout).
- Assign a StepperViewTemplate to the new View and name it ExperienceLevelStepper.
- The StepperViewTemplates properties are:
    - stateField: STATE
    - titleField: TITLE
    - iconField: ICON
- Set PersonExperienceLevelStepper_view for property lookupView of Context PersonExperienceLevel.

Now, go back to Person_entity and create a new Consumer named PersonExperienceLevelStepper. Its properties are:

- entityName: PersonExperienceLevel_entity
- fieldName: #PROVIDER
- state: EDITABLE

If you unfold this new Consumer, you will see its Parameters. Set the valueProcess of CurrentLevel_param as follows:

```
result.string(vars.get("$field.EXPERIENCELEVEL"));
```

Select Consumer PersonExperienceLevelStepper for property "consumer" of EntityField EXPERIENCELEVEL.

Finally, just for testing purposes, add EntityField EXPERIENCELEVEL

- to ViewTemplate "Edit" of PersonEdit_view and
- to ViewTemplate "Info" of PersonPreview_view

Now, deploy your changes, navigate to Context "Contact", and open any contact person in its MainView. Open tab "Experience Level", which includes the StepperViewTemplate. Test it by clicking on the pencil icon and then on a step other than the current one. Press "Save" to save the chosen step.

In the above example, Parameter DisabledLevels_param has not been used so far. Its purpose is to disable specific steps (here: experience levels), so they cannot be selected (which will be indicated by a darker icon color). The logic for this is done on Consumer side (Person_entity > Consumers > PersonExperienceLevelStepper > DisabledLevels_param), in the Parameter’s valueProcess. The result of this valueProcess must be a stringified array of the KEYIDs of all steps that should be disabled. Here is an example code that disables the steps BEGINNER and PRO:

```
result.string(JSON.stringify([$KeywordRegistry.ExperienceLevel$beginner(), $KeywordRegistry.
ExperienceLevel$pro()]));
```

Please be aware that the above implementation is only a plain example. Actually, it is possible to realize use cases that are far more complex. E.g., you are not forced to use keywords, and the handling of the states can be done more dynamically instead of using only one Parameter for disabling, etc.

11.3.25. Table

A ViewTemplate of type "Table" is used to show multiple fields of multiple datasets in a table. You can select arbitrary EntityFields in property "columns": Open this property’s editor and add fields using the plus ("+") button. Afterwards, you can change them via the fields' combo boxes. To remove a field, select it (checkbox) and press the minus ("-") button. You can change the order of the fields by selecting them and moving them up or down with the arrow up/down buttons.

This ViewTemplate is commonly used for Filter Views.

Example:
"Organisations", a ViewTemplate of OrganisationFilter_view.

Appearance in the client:
In the client, you can find it under Contact Management > Company. It shows an image columns, followed by several titled columns: name, customer code, language, status, email, phone, and address.

Configuration:
"Organisations" has several fields of Organisation_entity specified in property "columns": #IMAGE, NAME, CUSTOMERCODE, LANGUAGE, STATUS, STANDARD_EMAIL_COMMUNICATION, STANDARD_PHONE_COMMUNICATION, and ADDRESS_ID.

> This ViewTemplate’s property flag "hideContentSearch" controls whether or not the content search bar (Context filter) of the table is hidden (true; default) or displayed (false). The content search bar will only work, if paging is disabled, i.e., the corresponding RecordContainer’s property flag "isPageable" must be set to false in this case.

11.3.26. Timeline

A ViewTemplate of type "Timeline" displays up to 7 EntityFields of multiple datasets, ordered in a vertical timeline. All fields have fixed positions: On the left, a date (property "dateField") is shown, followed by an icon ("iconField"). In the middle, in vertical order, 3 further fields ("titleField", "descriptionField", and "subdescriptionField") are shown in different colors. On the right, another field is shown ("additionalInfoField"). On the top of the timeline, an "informationField" is displayed, which appears if you mark a timeline dataset.

Optionally, further properties can be set, e.g., for displaying/hiding the time, and for controlling the order and the maximum number of the datasets.

Example:
"ActivitiesTimeline", a ViewTemplate of ActivityFilter_view (Context "Activity").

Appearance in the client:
In the client, you can find it under Contact Management > Activity. Make sure that "Timeline View" is selected via the button in the View’s upper right corner.

In this case, the informationField, the subdescriptionField, and the additionalInfoField do not exist.

Configuration:
"ActivitiesTimeline" has 4 fields of Activity_entity specified: entryDateDateFormat (dateField), SUBJECT_DETAILS (titleField), INFO (descriptionField), #IMAGE (iconIdField). informationField, subdescriptionField, and additionalInfoField are not yet configured. You may set them arbitrarily to watch the effect in the client.

11.3.27. Tiles

Displays multiple datasets as "tiles". Each tile is styled like a business card (similar to template type "Card"), allowing to show up to 6 EntityField at fixed positions: On the left, an image (property "iconField"); on the right, 5 further fields (properties "titleField", "infoTopField", "subtitleField", "descriptionField", and "infoBottomField").

Property "tilePresentation" controls if the tiles are to be ordered as LANDSCAPE (default; 3 tiles in a row, dynamic width), or PORTRAIT (dynamic number of tiles per row, fixed width).

Example:
"Tiles", a ViewTemplate of ProductFilter_view (Context "Product")

11.3.28. TitledList

A ViewTemplate of type "TitledList" is used to show multiple fields of multiple datasets in a list. You can select arbitrary EntityFields in property "columns": Open this property’s editor and add fields using the plus ("+") button. Afterwards, you can change them via the fields' combo boxes. To remove a field, select it (checkbox) and press the minus ("-") button. You can change the order of the fields by selecting them and moving them up or down with the arrow up/down buttons.

Furthermore, in property "titleField" you can select one EntityField to be displayed as "title", to the left of the "columns" fields.

To improve performance the TitledList ViewTemplate by default only displays the first six rows and then shows a "+ X additional rows" label, which on clicking it loads the rest of the rows. The amount of rows shown can be configured using the property `rowLimit` of this ViewTemplate.

Example:
"Addresses", a ViewTemplate of AddressList_view.

Appearance in the client:
The "Addresses" ViewTemplate is shown, e.g., in the PreviewViews of Contexts Person and Organisation.

Configuration:

- In property "columns", the different parts of an address are specified, e.g., country, zip code, or city.
- Property "titleField" includes the EntityField ADDR_TYPE, which holds the title information.
- Property "highlightingField" is set to the EntityField IS_STANDARD, which makes standard addresses being highlighted (title is shown in bold font).

11.3.29. Tree

A ViewTemplate of type "Tree" is used to show, at fixed positions, specific fields of multiple datasets, grouped to a tree. To configure the tree structure shown by default, you can select arbitrary EntityFields as grouping criteria in property "defaultGroupFields": Open this property’s editor and add fields using the plus ("+") button. Afterwards, you can change them via the fields' combo boxes. To remove a field, select it (checkbox) and press the minus ("-") button. You can change the order of the grouping by selecting fields and moving them up or down with the arrow up/down buttons.

The "leafs" of the tree can be configured with the following fields: An icon on the left (property "iconField"), on the right a title ("titleField"), and below a description ("descriptionField").

Optionally, you can add up to 3 ActionGroup buttons by selecting an ActionGroup in field "favoriteActionGroup1", "favoriteActionGroup2", or "favoriteActionGroup3".

Each grouping node’s name can optionally be displayed extended by the number of sub-nodes, or leafs, respectively (property flag "showChildrenCount"). In the client, the configured default grouping can be modified via the filter window (click on "Filter" button).

Example:
"Treetable" (should better be named "Tree"), a ViewTemplate of 360DegreeFilter_view (included in the MainView of various Contexts, e.g., Context "Company").

Appearance in the client:
In the client, you can, e.g., find it under Contact Management > Company > Open the MainView of any company > 360 Degree. (If the tree is not shown, select "Tree View" via the button in the upper right corner.) The tree is grouped by the Contexts' names (e.g., "CONTRACT"). These grouping nodes can be expanded, in order to display their objects (e.g., all contracts of the company), each consisting of an icon, a title, and a date.

You can change the grouping by clicking on the "Filter" button and adding/removing grouping fields.

Configuration:
"Treetable" has the following fields of 360Degree_entity specified: CONTEXT_NAME (defaultGroupFields), ICON (iconField), TITLE (titleField), and DATE (descriptionField). Furthermore, the ActionGroup "newModule" is selected in property "favoriteActionGroup2". Property "showChildrenCount" is set to "true".

Find further details in chapter Tree and TreeTable: Advanced explanations below.

11.3.30. TreeTable

A ViewTemplate of type "TreeTable" is used to show multiple fields of multiple datasets in a table, which can be grouped to a tree. Rows are datasets, columns are EntityFields. You can select arbitrary EntityFields in property "columns": Open this property’s editor and add fields using the plus ("+") button. Afterwards, you can change them via the fields' combo boxes. To remove a field, select it (checkbox) and press the minus ("-") button. You can change the order of the fields by selecting them and moving them up or down with the arrow up/down buttons.

To configure the tree structure shown by default, you can select arbitrary EntityFields as grouping criteria in property "defaultGroupFields".

This ViewTemplate is commonly used for Filter Views.

Example:
"ActivitiesTreeTable", a ViewTemplate of ActivityFilter_view.

Appearance in the client:
In the client, you can find it under Contact Management > Activity. It shows a date column, followed by an image and several titled columns: responsible, subject, and description. In section "Grouping" of the filter component shown to the right of the table (if not visible, click button "Filter" first), you can group the activities by a specific criteria, e.g., by category.

Configuration:
"ActivitiesTreeTable" has several fields of Activity_entity specified in property "columns": entryDateDateFormat, #IMAGE, RESPONSIBLE, SUBJECT, INFO. That’s all.

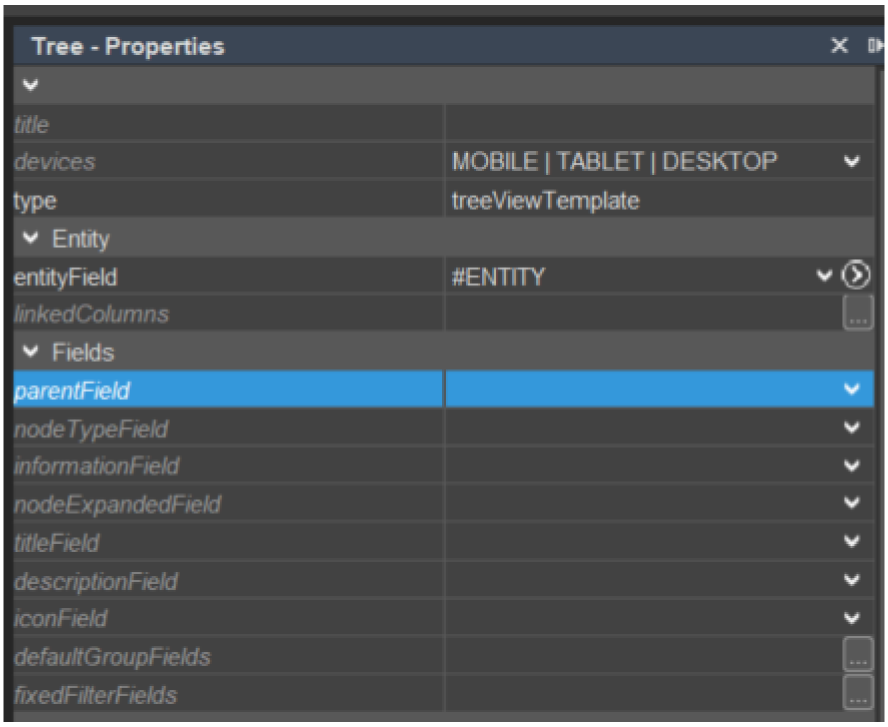
Other than in a Tree, a TreeTable can optionally feature "drag and drop". To enable this, you need to set property "enableDragAndDrop" to "true" (checkbox checked). In this case, you should integrate LexoRank as ordering algorithm (rather than a numeric ordering) - find detailed information in appendix LexoRank.

Find further details in chapter Tree and TreeTable: Advanced explanations below.

11.3.31. Tree and TreeTable: Advanced explanations

The ViewTemplates of type Tree and TreeTable are used to display connected data within a tree structure. Both ViewTemplates behave similar and only differ in their properties. The TreeTable is an advanced version of the Tree. As a datasource, a JDitoRecordContainer is used in most cases (see chapter JDitoRecordContainer). If the data is already stored correctly and does not need further manipulation, a DatabaseRecordContainer can be used instead.

11.3.31.1. Important properties - Tree


Tree - Properties

title	
devices	MOBILE | TABLET | DESKTOP
type	treeViewTemplate
Entity	
entityField	#ENTITY
linkedColumns	
Fields	
parentField	
nodeTypeField	
informationField	
nodeExpandedField	
titleField	
descriptionField	
iconField	
defaultGroupFields	
fixedFilterFields	

- entityField: Here we commonly set #ENTITY, because the data is taken from various EntityFields.
- linkedColumns: With this property, you can set EntityFields to be used to immediately open the dataset in the MainView.
- parentField: This is the central field to construct the tree structure. In this field, the ID of a parent node has to be hold, if the entry is a child node. More details follow below.
- informationField: This field contains information that is displayed on the top of the entry.
- nodeExpandedField: This field has to be of the type BOOLEAN and determines if the node is opened or closed at first. For example, if you want to have the tree completely expanded, then you just assign the value `true` to it, by either using its valueProcess (which is not recommended, as the valueProcess gets executed for each entry and thus can lead to a low performance) or by assigning the value in your RecordContainer.
- titleField: This is the first row of information you can use to display your data. It is called titleField, because here you would typically put the name of an entry, like the name of an organisation.
- descriptionField: This is the second row of information and can be used to display further information.
- iconField: With this field you can add an icon to the entry. Use either #IMAGE or #ICON, if you want to use the imageProcess or iconProcess of the Entity or you can use one of your EntityFields, which has to contain one of the following:

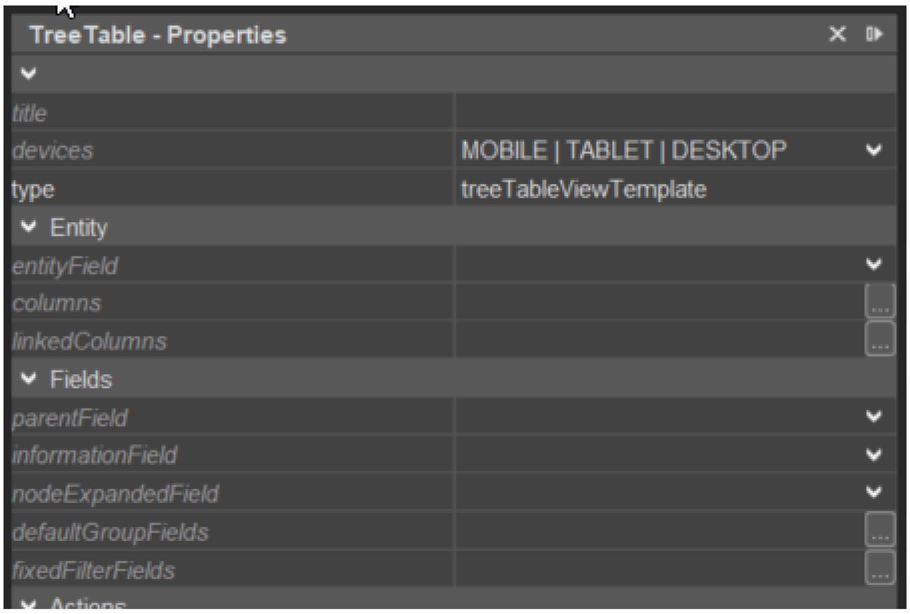
    - A text in the format "TEXT:" + your desired values. This text gets evaluated and turned into two initials and an automatically selected background color.
    - The name of one of our predefined icons, like "NEON:LOGO".
    - An image encoded in Base64. If you use an image of your own, always use a scalable vector graphic (.svg), so the image can automatically be scaled properly to meet different screen resolutions.

- defaultGroupFields: If your Entity has group fields, you can determine which ones are used by default.
- fixedFilterFields: With this property you can set the filter fields for the tree, so only those can be used.
- expandRootItems: Select if the root nodes are expanded by default or not. Set to true to expand all roots by default.

If your Entity doesn’t use a paging RecordContainer and the property is set to `true`, **all** items are loaded at once, which can lead to a loss in performance. Use this property carefully!

11.3.31.2. Important properties - TreeTable

Before reading this chapter, we recommend to read chapter "Important properties - Tree" first.


TreeTable - Properties

title	
devices	MOBILE | TABLET | DESKTOP
type	treeTableViewTemplate
Entity	
entityField	
columns	
linkedColumns	
Fields	
parentField	
informationField	
nodeExpandedField	
defaultGroupFields	
fixedFilterFields	
Actions	

- entityField: Here we commonly set #ENTITY, because the data is taken from the different EntityFields.

- columns: This is the property that distinguishes the TreeTable from the Tree. In this property, you can add multiple EntityFields as columns for each entry, which then are displayed in a table style.
- linkedColumns: This property has the same function as it has in the Tree ViewTemplate.
- parentField: This field holds the ID of the parent node, if the entry is a child node, otherwise it has to be null to mark this entry as a node of the first layer.
- informationField: A field for information to be displayed at the top of the entry.
- defaultGroupFields: If your Entity has group fields, you can determine which ones are used by default.
- fixedFilterFields: With this property you can set the filter fields for the tree, so only those can be used.
- expandRootItems: Select if the root nodes are expanded by default or not. Set to true to expand all roots by default.

If your Entity doesn't use a paging RecordContainer and the property is set to `true`, **all** items are loaded at once, which can lead to a loss in performance. Use this property carefully!

11.3.31.3. Building a Tree/TreeTable

If you are using a Database RecordContainer, set it up like you are used to and make sure to link your EntityFields correctly. But please note: If property isPageable is set to true, then the tree does not work with the parent-child principle that is explained below.

When using a JDitoRecordContainer, the approach gets a bit more complex as you have to do your data selection manually. The general rules for the JDitoRecordContainer apply (see chapter "JDitoRecordContainer"). Additionally, consider the following rule:

You have to ensure the correct order of the datasets. Parent nodes have to be added to the result array before any of their child nodes.
For example, if the tree structure looks like this:
A
- A1
- A2
B
- B1
- B2

Then A has to be added before A1 and A2, as well as B has to be added before B1 and B2

This rule also applies when using a Database RecordContainer and can lead to exceptions if you don’t order your datasets correctly.

There are two strategies that can be used to build your data:

1. Layer by layer
   This strategy aims to add one layer at a time. This can be done if you have different, but related sets of data. Example: The first layer contains organisations, the second layer contains persons with functions at those organisations, and the third layer contains all Activities linked to those persons.
2. Recursive
   The recursive strategy is used if your data can be infinitely deep and could branch out infinitely. To add your data to your result, you have to do two steps:
   a. Find all your nodes of the first layer and add them to your result.
   b. Iterate over all top nodes and use a recursive function to retrieve the children of the next level. The recursive function has to end, if no further children are found.
      By the way, this strategy is used in the 360° View, for example.

11.3.31.4. Examples

In the following text, we will go over multiple examples. As Tree and TreeTable behave in the same way, we will focus on only one of them in each example.

11.3.31.4.1. Simple Tree of organizations and their persons

This example will build a Tree that lists all organizations and the persons connected with them. It will not cover displaying their PreviewViews or doing anything with the data. The focus is just on building the tree.

First we build an Entity named "OrgTree_entity" and three EntityFields (UID, PARENT, TITLE). Then we add a JDitoRecordContainer and create a Context called "OrgTree". Furthermore, we add a View "OrgTree_view", assign "OrgTree_entity" to the Context and set the Context’s property filterView to "OrgTree_view". Then we open "OrgTree_view" in the Navigator window. Within the View, we add a Tree ViewTemplate and make sure that "#ENTITY" is assigned in the `entityField` property. Then we fill the `parentField` with our PARENT EntityField and `titleField` with the EntityField TITLE. Now add the Context to the Global Menu under application > ____SYSTEM_APPLICATION_NEON. This completes the basics for this example.

Now we go back to our Entity and open the `contentProcess` of the RecordContainer. Here we build the data to be displayed in the tree. First, we will gather our data, then we put it in the right format, and finally we return it. This example will use the "layer by layer" strategy (see above).

```
import { result } from "@aditosoftware/jdito-types";
import { newSelect } from "SqlBuilder_lib";

// First we get our two layers of data.
// It could be done with one SQL statement,
// but that would make the example more complex.
// We go for simplicity in this one.
// Getting person data, using the SqlBuilder
var personData = newSelect("CONTACTID, FIRSTNAME, LASTNAME,
ORGANISATION_ID")
    .from("PERSON")
    .join("CONTACT", "PERSONID = PERSON_ID")
    .where("ORGANISATION_ID is not null")
    .and("PERSON_ID is not null")
    .table();

// Getting organisation data, using the SqlBuilder
var orgData = newSelect("CONTACTID, ORGANISATIONID, NAME")
    .from("ORGANISATION")
    .join("CONTACT", "ORGANISATIONID = ORGANISATION_ID")
    .where("ORGANISATION_ID is not null")
    .and("PERSON_ID is null")
    .table();

// Now we prepare our result array
var res = [];

// For our UID field, we want to use the CONTACTID.
// As we have selected the ORGANISATIONID with both our layers,
// we need to replace the ORGANISATIONID in our person data
// by the organisation's CONTACTID.
for(let i = 0; i < orgData.length; i++)
{   for(let j = 0; j < personData.length; j++)
    {   if(personData[j][3] == orgData[i][1])
        {
            personData[j][3] = orgData[i][0];
        }
    }
}
// As the organisations are the first layer,
// we will add their data to the array
// The order of our fields will be: UID, PARENT, TITLE
for(let i = 0; i < orgData.length; i++)
{
    res.push([orgData[i][0], null, orgData[i][2]]);
}

// Now we add our second layer
for(let i = 0; i < personData.length; i++)
{
    res.push([personData[i][0], personData[i][3], personData[i][1] +
" " + personData[i][2]]);
}

// Last step: Returning the data to the system
// using result.object(), because we need
// to return an array, not a String.
result.object(res);
```

After that is done, we open the dialog of the `recordFieldMappings` property of the RecordContainer and add UID.value, PARENT.value, and TITLE.value.

Now we can deploy our changes and test it in the web client.

This chapter will be extended in a future version of this manual.

11.3.32. WebContent (IFrame)

Displays a web page, with the URL or HTML content being defined in an EntityField (specified in the ViewTemplate’s property "entityField"). The dimensions of the component’s appearance in the client can be defined using properties width, height, and unit.

Example:
ViewTemplate "Timeline" of View "FacebookTimeline_view" (of Context "Social"). This ViewTemplate’s property "entityField" references the EntityField FACEBOOK_TIMELINE (of Social_entity), which has a valueProcess, whose result is HTML code defining the web content to be displayed. The web content is visible in the client, when you add the Dashlet "ADITO Facebook Feed" (in the DashletStore’s Dashlet group "Social Media") to your Dashboard.

11.3.32.1. Advanced explanations

The ViewTemplate "WebContent" is controlled by the properties of

- the ViewTemplate itself
- the EntityField that is referenced in the ViewTemplate’s property "entityField"

The settings differ depending on the content to be displayed:

- Content source
  The source of the content is always the valueProcess of the EntityField. Example:

  valueProcess of EntityField TWITTER_TIMELINE of Social_entity

  ```
  import { vars } from "@aditosoftware/jdito-types";
  import { result } from "@aditosoftware/jdito-types";

  var account = vars.get("$param.Account_param");
  result.string("<html onmouseover=this.className='scroll' onmouseout=this.className='noscroll'
  class='noscroll'><head><style>.scroll { overflow: auto; }.noscroll { overflow: hidden; }</style></head><body><a class=
  \"twitter-timeline\" href=\"https://twitter.com/"+account+"?ref_src=twsrc%5Etfw\">Tweets by "+account+"</a> <script async
  src=\"https://platform.twitter.com/widgets.js\" charset=\"utf-8\"></script></body></html>")
  ```

- Loading URLs
  If the EntityField's property contentType is set to LINK, the URL will be used that is given in the valueProcess, and the web page will be loaded in the IFrame.

- Loading HTML
  If the EntityField's property contentType is set to HTML, the HTML will be used that is given in property valueProcess, and it will be rendered in the IFrame.

- Heigth and width
  - If the ViewTemplate's property height is set to a specific value, then the height of the IFrame will be set to this value.
  - If the ViewTemplate's property width is set to a specific value, then the width of the IFrame will be set to this value.

- Unit
  Via the ViewTemplates property UNIT, the unit of height and width can be specified. You can choose between pixel and percent. If the unit is not set explicitely, pixel will be used as default.

- States
  Via the Entity's property "state"/"stateProcess", the state of the component can be controlled:
  - The IFrame does not change when READONLY or EDITABLE is set, because this component is always readonly.
  - The IFrame cannot be disabled (state DISABLED), because this component is always enabled.
  - If the state is set to AUTO, the IFrame is VISIBLE by default. If the IFrame should be invisible, the state must be set to INVISIBLE.

11.3.32.2. Common mistakes

Here are some common mistakes when using the WebContent ViewTemplate. We use the ADITO homepage as example: https://www.adito.de/

If you want to embed a hyperlink, then the content type must be set to LINK, and the valueProcess must provide the URL of the web page to be shown. A common mistake is to set the contentType to TEXT or HTML, and the valueProcess provides an IFrame, something like

```
<html><body><iframe src="https://www.adito.de/"/></body></html>
```

As the WebContent ViewTemplate itself is an IFrame, this code leads to nested IFrames, which results in suboptimal usage of space and to possible errors in display.

A further bad example refers to links in a custom HTML page. In this case, the right contentType is TEXT or HTML, both work. However, if you add a link like this to the page

```
<a href="https://www.adito.de/">ADITO Homepage</a>
```

then the page will open in the IFrame. If you want to avoid this, you need to add further keywords, such as

```
<a href="https://www.adito.de/" target="_blank" rel="noopener
noreferrer">ADITO Homepage</a>
```

Then, the web page will open separately.

11.3.33. Further ViewTemplate types

ViewTemplates of further types may be available in future ADITO versions. If you can find no ViewTemplate suitable for your purpose, please issue a request to ADITO via the Service Client.

11.4. Renderers

A Renderer is an ADITO model that can be assigned to an EntityField in a ViewTemplate (e.g., MultiEditTable), in order to change its appearance or its functionality (e.g., edit options).

Currently, there are 2 categories of Renderers available

- ViewRenderer: Renderer for changing the appearance.
- EditRenderer: Renderer for adding edit functionality (visible, e.g., as additional buttons).

To create a Renderer, navigate to "renderer" in the "Projects" window. Then, right-click on "renderer" and choose option "New" from the context menu. A model create dialog will appear, in which you enter the name of the renderer and leave the model type preset to "renderer". After confirming with "OK", a second dialog appears, requesting the Renderer's type.

Currently, the following Renderer types are available:

- NUMBERFIELD
- BADGE
- MULTISELECTCOMBOBOX

If a Renderer suitable for your use case already exists, you can re-use it, without having to create a new one. Each Renderer can be used for multiple use cases.

11.4.1. NUMBERFIELD

If you choose Renderer type NUMBERFIELD, an EditRenderer will be created. If assigned to an EntityField in property editRendererMapping of a ViewTemplate (only available in ViewTemplates of some types, e.g., MultiEditTable), the following additional buttons will appear in the client:

- Reset button (circular arrow): If you have changed an EntityField's value, but not saved yet, this button allows you to reload its value from the RecordContainer.
- "Set maximum" button: This button overwrites the currently displayed value with the maximum value as defined in the corresponding EntityField's properties "maxValue" or "maxValueProcess". If these properties are not set, the "Set maximum" button is not displayed.
- Incrementer and decrementer buttons:
    - By default, a "Plus" and a "Minus" button are displayed, allowing the client user to increase or decrease the currently displayed value in steps of 1.
    - You can customize the step by setting one of the following properties

- numberfieldStep: Enter a positive decimal value to define the step size. This will effect both the button for increasing and for decreasing the value. This property is ignored, if property numberfieldStepsProcess (see below) is set.
- numberfieldStepsProcess: Create arbitrary incrementer/decrementer buttons by defining their corresponding step size, which can be positive and negative decimal values, specified as an array. The following example results in four incrementer/decrementer buttons, corresponding to steps of size -100, -0.5, 0.5, and 100 (make sure to use `result.object` and not `result.string`, in this case):

  Example code for property numberfieldStepsProcess

  ```
  result.object([-100, -0.5, 0.5, 100]);
  ```

Example in xRM:
You can study an example of the usage of a Renderer of type NUMBERFIELD in ViewTemplate "MultiEditTable", included in ProductpriceFilter_view. Here, the Productprice_entity's EntityField PRICE has the Renderer "numberInput" assigned, see property "editRendererMapping". The Renderer "numberInput" itself can be found in the "Projects" window, under "renderer". For testing purposes, you may change its properties numberfieldStep or numberfieldStepsProcess, or set the EntityField PRICE's properties maxValue or maxValueProcess, in order to get familiar with the effects of this Renderer in the client. (Note that if there is code set for property maxValueProcess, then the value of property maxValue will be ignored.)

11.4.2. BADGE

If you choose Renderer type BADGE, a ViewRenderer will be created. If assigned to an EntityField in a ViewTemplate (e.g., of type MultiEditTable), the EntityField's value will be displayed on a background that shows the color that is defined in the EntityField's properties color or colorProcess. (The font color will automatically be shown in the complementary color.) Renderer type BADGE has no specific renderer properties to set, it is simply created and assigned to an EntityField using the ViewTemplate's property viewRendererMapping (only available in ViewTemplates of some types, e.g., MultiEditTable).

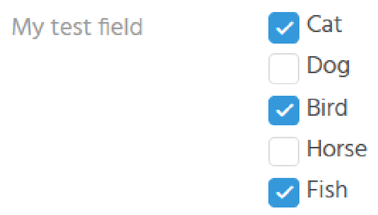
Example in xRM:
You can study an example of the usage of a Renderer of type BADGE in ViewTemplate "MultiEditTable", included in ProductpriceFilter_view. Here, the Productprice_entity's EntityField PRICELIST has the Renderer "badge" assigned, see property "viewRendererMapping". The Renderer "badge" itself can be found in the "Projects" window, under "renderer". For testing purposes, you may edit the EntityField PRICELIST's properties color or colorProcess, in order to get familiar with the effects of this Renderer in the client. (Note that if there is code set for property colorProcess, then the value of property color will be ignored.)

11.4.3. MULTISELECTCOMBOBOX

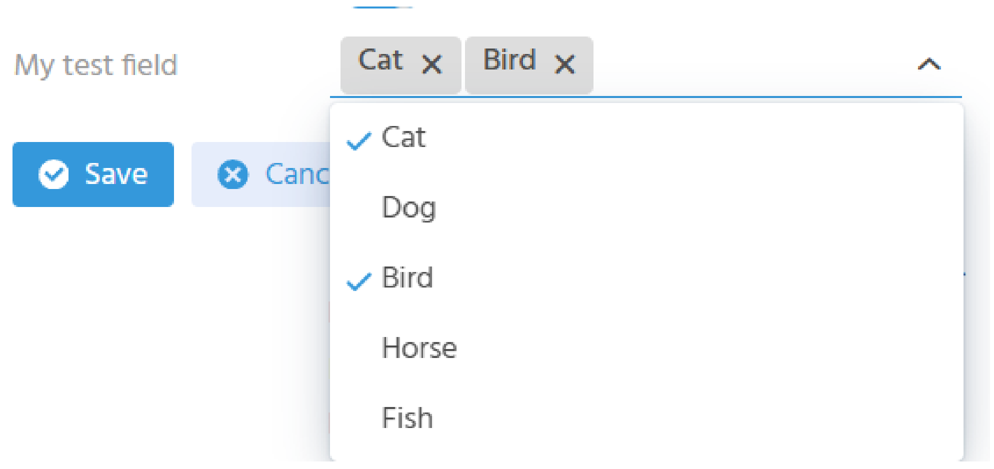
Using a Renderer of type MULTISELECTCOMBOBOX is an alternative to using a list of checkbox items that are all visible permanently.

11.4.3.1. Basics

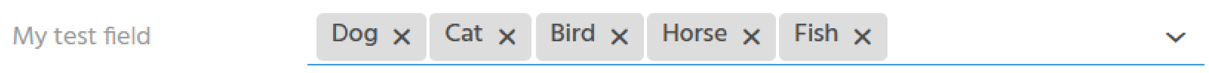
Given an EntityField that holds multiple items (via dropdownProcess or via Consumer), with the option to select one or multiple of them (selectionMode = MULTI). By default, this will result in a list of all selectable checkbox items. Now, for a small number of items, this is fine:



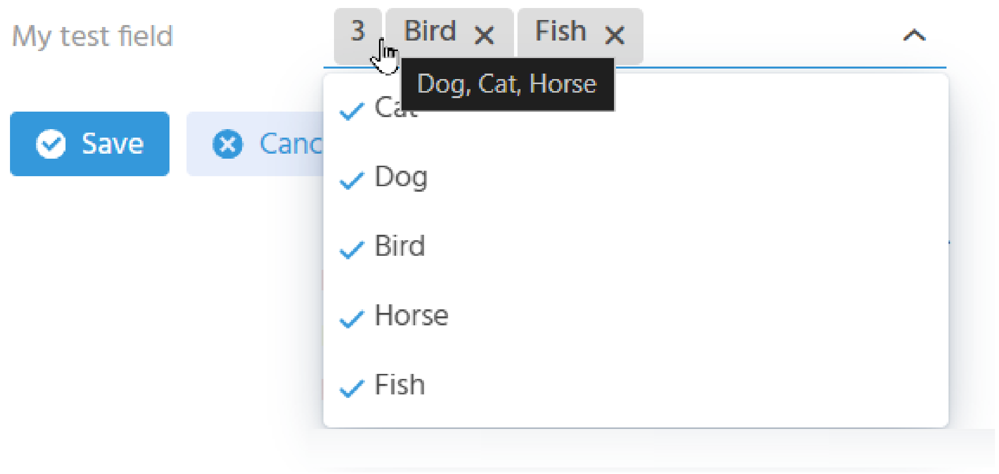
However, a larger number of selectable items might spoil the respective ViewTemplate's appearance. In these cases, the Renderer MULTISELECTCOMBOBOX should be preferred. It is an EditRenderer and packs all checkbox items in a combo box. Only if you open the combo box, the selectable items are visible (if required, in a scrollable way).



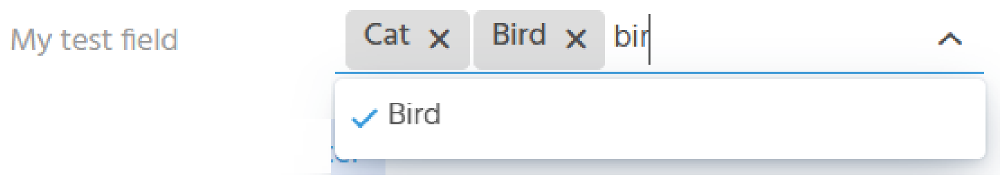
Independent from if the combo box is opened or closed, the selected items (or a part of them) are shown above the combo box, along with the option to deselect them (via a cross icon to the right), without having to open the combo box.



How many of the selected items are shown above the combo box depends on the available space. If there is not enough space to display *all* selected items, then a number is shown on the left. Example: If, e.g., 5 items have been selected in total, with the last 2 of them shown above the combo box, then this number is 3. If you hover over this number with the mouse pointer, the further selected items are shown in a tooltip.



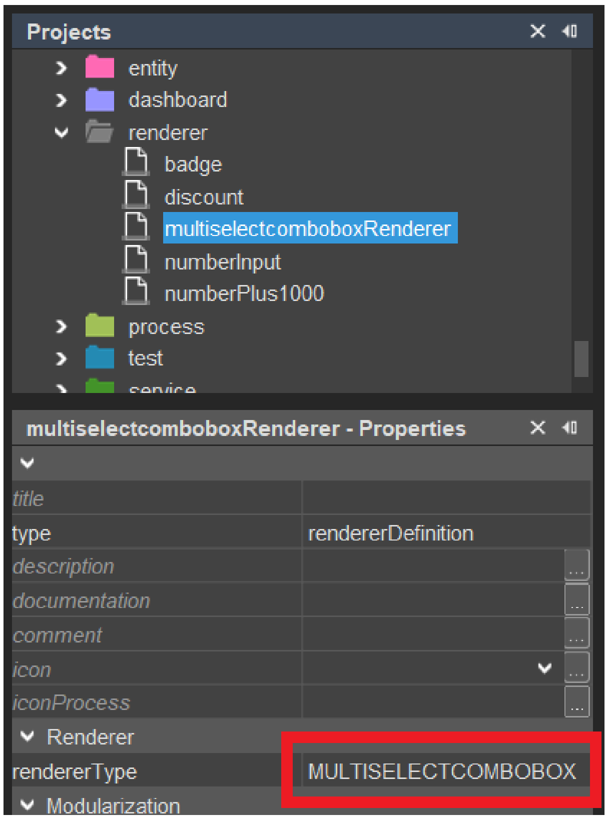
Furthermore, the MULTISELECTCOMBOBOX includes a filter: Simply type in a filter value on the right of the selected items, then the combobox will be filtered accordingly:



11.4.3.2. Configuration

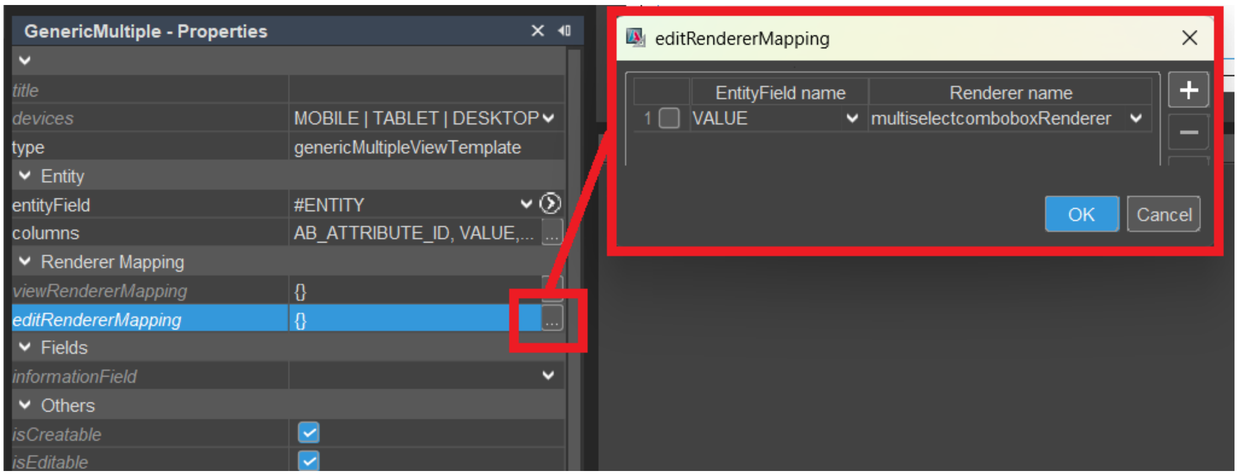
The configuration of a MULTISELECTCOMBOBOX takes 2 steps:

1. Create a new Renderer of type MULTISELECTCOMBOBOX
   In the "Projects" window, right-click on folder "renderer" and choose "New" from the context menu. In the following dialog, give the new Renderer a suitable name and make sure that "renderer" is selected as type. After clicking "OK", select MULTISELECTCOMBOBOX in the subsequent dialog. Once confirmed with "OK", the new Renderer is added to the list of existing Renderers.



title	
type	rendererDefinition
description	
documentation	
comment	
icon	
iconProcess	
Renderer	
rendererType	MULTISELECTCOMBOBOX
Modularization	

2. Now, the new Renderer can be assigned to the respective multiselection EntityField, via property editRendererMapping of a ViewTemplate (only available in ViewTemplates of some types, e.g., Generic):



title	
devices	MOBILE | TABLET | DESKTOP
type	genericMultipleViewTemplate
Entity	
entityField	#ENTITY
columns	AB_ATTRIBUTE_ID, VALUE,...
Renderer Mapping	
viewRendererMapping	{}
editRendererMapping	{}
Fields	
informationField	
Others	
isCreatable	✓
isEditable	✓

	EntityField name	Renderer name
1	VALUE	multiselectcomboboxRenderer

11.4.3.3. Value format

When working with an EntityField that has a Renderer of type MULTISELECTCOMBOBOX, the value format is the same as it is without the Renderer.

Example:

Given an EntityField MYTESTFIELD, with selectionMode = MULTI and a dropDownProcess as follows:

```
result.object([
    ["id1", translate.text("Cat")],
    ["id2", translate.text("Dog")],
    ["id3", translate.text("Bird")],
    ["id4", translate.text("Horse")],
    ["id5", translate.text("Fish")]
]);
```

If you select and save, e.g., "Cat", "Bird", and "Fish", then

- `vars.get("$field.MYTESTFIELD")` as well as
- `vars.get("$this.value")`

will return a multistring - in this case `"; id1; id3; id5;"`.

Therefore, if you want to further process the value (e.g., in an onDBInsert process), you will have to decode it first, in order to get it as an array:

```
var myFieldValue = vars.get("$field.MYTESTFIELD") ; // ";id1; id3; id5;"
var myFieldValueArray = text.decodeMS(myFieldValue); // ["id1", "id3", "id5"];
```

Vice versa, if you want to preset the selection of specific items of the combo box, in a valueProcess, then you need to encode the value first:

```
if (vars.get("$sys.recordstate") == neon.OPERATINGSTATE_NEW && vars.get("$this.value") == null) {
    var initialValue = ["id1", "id2"];
    var encodedValue = text.encodeMS(initialValue);// "; id1; id2; "
    result.string(encodedValue);
}
```

11.5. Device-specific designs

ADITO includes an automatic optimization of its design, depending of the type of device on which it is used:

- Desktop
- Tablet
- Mobile

Several ADITO models (e.g., Contexts or ViewTemplates) have a property named "devices", which allows you to control, on which of the 3 device types ADITO will be available.

For demonstration or testing purposes, you can also show the tablet-specific and the mobile-specific designs in your (Chromium-based) desktop browser.

Here is an example of showing the mobile-specific design in a Google Chrome or Microsoft Edge browser:

1. Log out
2. Close the browser tab in which ADITO had run
3. Enter the basic URL of your ADITO system, adding the suffix "/mobile", e.g., https://myProject.dev.c2.adito.cloud/mobile
4. Click key "F12"
5. A developer window will appear. Here, click subsequently on
    a. the three-dotted button (upper right corner) and choose "Dock side" > "Undock…" button
    b. "Toggle Device toolbar" button (left upper corner of the developer window)
6. Click key "F11"

If you want the tablet-specific design to be shown, proceed as described above, but use the suffix "/tablet" instead of "/mobile".

If you are done with testing, subsequently click the keys "F11" and "F12" again, in order to return to your normal browser usage.

Due to browser "cookies", you might have problems to return to the "normal" (desktop-specific) design, if you only enter the usual URL (e.g., https://myProject.dev.c2.adito.cloud/ or https://myProject.dev.c2.adito.cloud/client). This possibly still opens ADITO showing the tablet- or mobile-specific design (= the latest design that you had used). In these cases, repeat the above steps, using the suffix "/desktop".

11.6. Further design elements

11.6.1. Icons

Well-selected icons help the user to navigate through the client and its menu items. ADITO provides you with a large number of predefined icons, which you can assign to various components, such as menu groups, or Contexts. Besides, you can also use any icon from your own resources. If no icon is selected, a question mark ("?") is displayed instead.

11.6.1.1. Predefined icons

Whenever an ADITO component shows a property named "icon" or "iconId", you can use the combo box to select from a long list of predefined icons. Then deploy. Whenever an icon is included in a menu, you need to log out and log into the web client first, in order to see the icon.

Example:
As you will have noticed, menu group "Car Pool" is currently displayed with a question mark to the left of it. To replace it by an icon double-click on application > _SYSTEM_APPLICATION_NEON, and then, in the visual representation of the Global Menu (menu editor in the Editor window), click on "Car Pool". Then, in the properties window, select a suitable icon, e.g., "VAADIN:CAR".

You can now repeat these steps in order to assign icons also to the car pool-related Entities (not to the Contexts, whose icons are only shown in the Designer). These icons will then, e.g., appear on the top of a Context (in blue font color) and in the vertical navigation bar to the very left of the client (to be opened via the "burger button" in the upper left corner of the client).

Entities have multiple icon-related properties. Please ignore the outdated properties "icon" and "iconProcess", and use (further below in the property sheet, in section "Icon") the properties "iconId" and "iconIdProcess" instead.

You can quickly find a suitable icon, if you filter the "icon" combo box using the starlet ("*") as preceding wildcard.
Example: You are looking for an icon that has something to do with a "circle". To find all possibly suiting icons, proceed as follows:

1. Open the "icon" combo box by clicking on the small "arrow" to the left of the 3-points button: A long list of all icon names, along with a preview of all icons, will appear.
2. Type the following string: *circle
   → The focus will jump to the first icon with its name containing the word "circle"
3. Use the arrow keys (down/up) to navigate to all other icons having names containing "circle". (This is better than scanning the whole list using the scroll bar.)
4. Press the "Enter" key to select the icon of your choice.

11.6.1.2. Icons from user's resources

You can use icons from your own resources also via property "icon" - but now, do not open the combo box, but click on the three-dotted edit button ("…"). This will open a file selection dialog; here, you can navigate to the folder, in which you have stored the icon, and select ("Open") it there. Then, instead of the file’s name, only the placeholder "[BINDATA]" is shown as property value. Note that the icon will not be stored as a file in the ADITO project, but it will be inserted as binary value in the system database table ASYS_BINARIES.

> All parts of the ADITO xRM project are already equipped with well-suited icons. This ensures a consistent layout of the ADITO client. Therefore, it is strictly against the intention of ADITO that these icons are replaced by user-defined icons. Exceptions are possible, if an icon represents a specific system (e.g., a connected ERP system), a product, or a company.

Generally, ADITO recommends to use exclusively vector graphics, included in SVG files. This allows an arbitrary scaling/zooming, and the programmer does not have to consider the resolution of the image.

If you nevertheless want to use absolute icons (like those of format BMP, PNG, or JPG), you need to design them according to purpose, display resolution, and zoom level (an icon appearing great on a 24“ HD display may look bad on a 4K display). Therefore, ADITO recommends to use only vector graphics as icons.

> As the client’s elements can be zoomed in the browser, there is no absolute size of icons. Currently, icons appear smallest in the Global Menu, and largest in Views including "tiles".

11.6.1.3. Variable icons

If you want to use various, logic-dependent icons, enter the required code in property "iconProcess". All icons must at first be stored in the system database table ASYS_BINARIES. The result of the iconProcess (`result.string(…)`) must be the value of field ASYS_BINARIES.ID of the dataset representing an icon.

11.6.1.4. Avatars

In ADITO, so-called Avatars are tiny visual components that are auto-generated by the ADITO platform’s logic for all EntityFields of contentType IMAGE.

Example:
In PersonFilter_view, column PICTURE shows icon-typed images of the persons. However, if a Person dataset does not include an image (EntityField PICTURE), the first letters of first name and last name are displayed on a square, colored background. (The configuration of this behavior is done in PICTURE’s displayValueProcess, see below.)
Both variants - image and letter-based substitution - are named "Avatars" in ADITO.

> For a better understanding of the following, you need to know that the ADITO platform automatically converts binary images (after being loaded from the database) into base64 strings, before perfoming subsequent logic. Thus, the value of an EntityField of contentType IMAGE is always a string, not a binary object.

The ADITO platform’s Avatar auto-generation logic for EntityFields of contentType IMAGE works as follows:

- If the EntityField is to be displayed in a ViewTemplate of type "Picture", the value is interpreted as base64 image, which is then shown without additions (no background etc.). The following steps are skipped.
- The logic checks if the value includes character ":" (colon). If not, the value is interpreted as base64 image, which is then shown it without additions (no background etc.). The following steps are skipped.
- The subsequent steps of the logic depend on the prefix (= the string part preceeding the ":" character):
    - prefix "TEXT": 0-2 characters are shown on a colored, square background, according to the following logic:
      First, the part of the string that follows the ":" character is split into substrings, using " " (space) as delimiter.
      Empty substrings are sorted out. The remaining substrings are the actual basis for generating Avatar **character(s)**:
        - If there is only one substring, its first 2 characters are used. If the substring contains only 1 character, the Avatar will show only this single character.
        - If there are multiple substrings, the first characters of the first 2 substrings are used. There is one exception: If the first character of a substring is a non-letter character, then this substring is skipped, and instead the first letter of the next

substring is used. This logic is helpful when, e.g., a company name like "Energystar - Oil Company" is given; in this example, the letters "EO" would be extracted, not "E-".

    - Example from the xRM project’s Context "Contact": If there is no image of the person available, the displayValue of EntityField PICTURE starts with "TEXT:", followed by the values of first name, last name, and company name (separated by a "space"). Thus, the auto-generated Avatar characters are usually the first letter of the first name and the first letter of the last name. If, however, a first name is not available (= this substring is empty), then the Avatar’s characters will be the first letter of the last name and the first letter of the company name.
    - The **color** of the Avatar’s square background depends on the color-related properties of the respective EntityField ("color" or "colorProcess"). If none of these properties is set, the background color is generated on the basis of a hash value of the original string.
      Note: The auto-generation logic is, of course, limited to the colors defined in the "Theme" (see chapters Themes and Color). Thus, the logic cannot ensure that a unique color is assigned to each single dataset.
  - prefix "NEON" or prefix "VAADIN": The complete string is interpreted as name of one of the official icons provided by the ADITO platform. You can browse the names of the available icons, e.g., via scanning the combo box of an "icon" property (see chapter Icons for further information).
  - prefix "URL": The part of the string that follows the ":" character is interpreted as an URL, pointing to an external image.

Example task:
Given be an EntityField MYIMAGEFIELD, which has the contentType IMAGE. Now, if no image can be loaded from the database, the value of MYIMAGEFIELD should be a combination of the first characters of text-typed EntityFields ENTITIYFIELD1 and ENTITIYFIELD2. If ENTITIYFIELD2 is empty or starts with a non-letter character, the first character of ENTITIYFIELD3 should be used instead.

To achieve this, you need to program a logic for setting a textual value for covering the case that a real image is not available. Usually, this is done in MYIMAGEFIELD’s valueProcess or displayValueProcess, including a case like this:

XXX_entity.MYIMAGEFIELD.displayValueProcess

```
(...)
if (vars.get("$field.MYIMAGEFIELD"))
    result.string(vars.get("$field.MYIMAGEFIELD"));
else
```

```
    result.string("TEXT:"
    + vars.getString("$field.ENTITIYFIELD1") + " "
    + vars.getString("$field.ENTITIYFIELD2") + " "
    + vars.getString("$field.ENTITIYFIELD2"));
(...)
```

The ADITO platform’s logic will then auto-generate an Avatar on the basis of the string as concatinated in the "else" part of the above code fragment.

Further examples:

For testing purposes, you may

- create an EntityField (e.g., named "PICTURE") for a suitable Entity of the xRM Project (e.g., Interest_entity),
- assign it to a suitable View (e.g., to InterestFilter_view), and
- enter the following valueProcess:

Interest_entity.PICTURE.valueProcess

```
// Avatar with characters "XA" on colored background
result.string("TEXT:xyz ABC mno");
//
// Icon showing "plus" symbol
// result.string("NEON:PLUS");
//
// Icon showing "factory" symbol
// result.string("VAADIN:FACTORY");
//
// URL pointing to image of Heinz Boesl (Founder of ADITO)
//
result.string("URL:https://www.adito.de/fileadmin/uploads/unternehme
n/team/heinz-boesl.jpg");
//
// ADITO logo as base64
//
result.string("UklGRroFAABXRUJQVlA4TK0FAAAvlUAJEE4hyLZZ/bWfM0TEBFT2B
5Hk2rbrtjkteOip8ysgnWatY6eoVBURD/AHGF8sCwmbSJKEdNX7T1/DZyiAHB/MFTLcR
JJtK8fcO2OA6GX8QuxR/XCDE0kAAJaRlLVt27ZtvWzbtm3btm3btm2fUQduIymSa0vHd
8twT4AYyVbYZvsvO8A3TgVfkh9qeY8kQQBAsM3FtW3btm3btm3btm3N3qvWbzZjO3HgN
pIiJceYmVn4Q8oIhowLJP+2DAtKVhMRlpe9t4ysh4QKtCSCLTNrIrLyuSaSeOv+/Lk/Z
MwaUVaU5f1OWVPSFEzWECGU1YsX1+DjfQrRPlLQOedQuxRN1hDRkB8dhwrKEiIwkrt7q
H2KLiuIUMhe16FCs4KIrvzuow4pBgyYPwInlZfWDhWR+SN0ctp6fE4h2qcWdUwxZvqIp
fxrPLuGnq47M+1B61BRGDB7BFGam24Z+bW+KMxGF6gTijmTR9jlpiWDf/wR+Q9m6rQYq
FgMmDzi0nZLakLFFKhTihUDpo6gSl9b+S4qCcVS3VDxGDB1hF+eWrImhEmouAJ1TrFn5
khg8VjSk+STOisGKmnmUxbBlIm28kPUks4NdUlxDo8rCIFZsKIE1/VB/Adw7sE5KnoK/
Wz/ZQETIkedLn0GDBn9TocGecIg6zhEUt5bsiHElxGh2KobKnX4Rp4bV0O3moDWErXvq
XTpwpnX1pXSHD3K3r88lc3OmavP3twbVJANNz77Aw75QF3X80D464s7e2JxzeOQmOKxX
EiNBAjibUyo+AJ1RXEPlrJ3S5c/jNcU09nQR77dbShJjPeNtpGFKaLlI2zUoGgEThLBl
6VGdWZOsVHnxchR6RTsGI1Fl3Ggb4g7shWoOwgj5d1WrN/MkkSU5VtX52XqmuIdKnWfk
qD5gZLEP+YlWBho6LOYU/QjpgRrNslS8Ajh9kuSFw0Jfoj4Yk2x8jEgiWSU3c8M/ThvO
majqXQq7VApVAv1pxiorKEb7SPIc1r2k8Q0cAWaBr4CJgWK1ul/kuguICNG3DlHrHLoJ
Ahqp4hDRuZogtXBnyMb1u8bDnw1gI8QwxiQCKlsXYycq7RrLMU/UDq+JUEW1jzgkoCzq
kDmWsVpSBLLAAvF6vqdBGrdSdQkiVWQohXAcJLYSiKa8uuiOq/rOlQOBdctgL6R4xm+x
mk3ScyDbMLtJLEDtAnD7moDVdqVEsdLjjEgAislAr0adUcJdjH0CztBEd7dPMcdJLtNm
MEkcQD8JdO0SutKQWOL9SgRGjl6vUPl924AjU7v4+9unPamzQN0D0ZQ/0DrTxIpXckbL
YnHiZjI344X19AQ6p4S7mDm77gX3cDP03wRkkBz3IPpuLH77RM9SSK1K6ejqYcRBKlvP
yv/ZFzKpeIu1b/9fB/eGz/t04BDFVHw5Qg/0wERyRN79n+/18PxNNbB9l+2BvAlWpNEY
VeKwUziKMIsly15EN4IzDnhi5g2kIq+hoqkhgrgrMpg1uAbswXDdQdcBkYRlZdXUpkkt
nEXsaD/byNC1UTs5X9Levv/t1Ci1FMxueaqu2MpO9rGOYNuVZOWSW1GvEQaQVTga0kbT
cgOVUiXKE5sS5Cm1MblwEelCqp0Ft8pbt2GQqBqiqmFYFPUcIQao/be/ap3aNgJHM/dM
x0ypfZaXKcpK84ycwP1m7IVIblrK3fC2RdKmfo8qxbXIyyYjSRxToUQ4dlQ8HaE2Me1i
WcUzmuRcLMVSx1TbBJbPJZOQRpAoVJdxRhCse/uXo+MGDLLOnGoSRD4b4HkKX0MGOBPT
Qvzc6OWeqLsIhsCyb9r5L+4ZySUMfUbcxHb9HwWBlDjyvnPhX3SZRxRWy6d/45GQNXnV
+nUW49s6RFGZHRRXn/H5x/l144leY9cOL+9oFYcc9kYd+qL/7BIKKj3/Ws4cOqcWqRKK
CUKHgMAAA==");
```

If the Avatar logic fails for whatever reason (e.g., the prefix or the base64 is not valid) a circled "exclamation mark" icon is shown instead of the Avatar (or, in some cases, no Avatar at all).

As an exercise, try to write a displayValueProcess for Car_entity’s EntityField PICTURE, in order to display a green Avatar, if an image of the car is not available.

11.6.1.5. Using gif files

Technically, you can also use an animated gif file instead of a static image, e.g., as picture of a contact person, or as background of the client’s login page. However, we recommend to use this feature with great care (if at all), as animations generally tend to distract the user’s attention from the actual information content shown in the client - and, in most cases, they are of no real additional benefit for the user.

11.6.2. Client navigation helpers

In ADITO, there are several ways to navigate through the various Contexts and Views. In the following chapters, you can find a description of additional functions, which increase the convenience when navigating in the client.

11.6.2.1. QuickEntry

The round blue button in the upper right corner of the ADITO client provides "QuickEntry" functionality, i.e., it enables you to quickly create new datasets of various Contexts. Technically, this means that specific EditViews are linked here. To add an EditView to the QuickEntry, proceed as follows:

- Click on the EditView in the "Projects" window.
- Set the View’s property "title". This text will appear in the list behind the quick create button.
- Open the editor dialog "Quick Entry" by clicking on the editor button (three-dotted button) in the View’s property "quickEntry".
- In this dialog, press the "Plus" button, to add a new line.
- Select the required EditView in the combo box of the new line.
- Move the new View up to the required position.
- Set a suitable icon in the EditView’s icon property. Otherwise, a question mark will be displayed in the QuickEntry list. You will need to log out and back in to make this take effect.
- Confirm with OK.

Do not set the "quickEntry" property for Views other than EditViews. This can lead to errors in the client.

To remove a View from the list behind the Quick Create button, right-click on its property "quickEntry" and choose "Restore Default Value" from the context menu.

11.6.2.2. linkedContext

Every Entity-Field has got a property named "linkedContext". Here, you can specify any Context of your application. The effect in the client is, that (not in all Views, but, e.g., in the PreviewView)

- beside the field, a small blue "eye" icon is shown. If you click on it, the PreviewView of the specified Context is opened.
- the field’s content is shown in blue font color, indicating a hyperlink. If you click on it, the MainView of the specified Context is opened.

11.6.3. Color

Colors help the user to navigate in the ADITO client. For example, a blue button with a white plus sign means "Here, you can create something new." Therefore, the definition of further colorings should always be done with caution and integrated into a general color concept.

Several ADITO models have color-related properties:

- color: Here, you can select a fixed value from a list of predefined colors. It is not possible to choose random colors.
- colorProcess: Here, you can enter code which ends with a `result.string()` command having a color value as argument. It is not possible to choose random colors, but you have to use one of the color values available via the constants `neon.`<…>`COLOR` (import "system.neon" first). This will prevent that, by mistake, colors are used that violate the defined "Theme" (see chapter Themes, subchapter of chapter Controlling the design).

> Currently, the color properties of an EntityField have only an effect in a limited number of cases, e.g., for
>
> - EntityFields that are displayed as colored score card (ViewTemplate type "Score Card")
> - EntityFields of contentType IMAGE (and there it works only if its valueProcess delivers one of ADITO’s predefined vector graphic icons, see example below). Furthermore, the number of available colors is also cautiously limited to those colors that are available via the constants `neon.`<…>, e.g. `neon.PRIORITY_HIGH_COLOR`. And, as you can see, these color constants do not refer to explicit color names, but they refer to their purpose.
> - specific types of Avatars (see chapter Avatars)

For exercise purposes, let’s introduce an additional column showing an icon, whose color depends on the color of the car:

- Add a new EntityField named COLOR_ICON to Car_entity
- Set the properties of the new field as follows:
  - contentType: IMAGE
  - valueProcess:

    Car_entity.COLOR_ICON.valueProcess

    ```
    var statusNew = $KeywordRegistry.taskStatus$new;
    ```

```
result.string(TaskUtils.getStatusIcon(statusNew));
```

(We simply "borrow" a circle-type icon from Context "Task".)

- colorProcess:

  Car_entity.COLOR_ICON.colorProcess

  ```
  switch(vars.get("$field.COLOR"))
  {
      case $KeywordRegistryCarPool.carColor$green():
          result.string(neon.PRIORITY_LOW_COLOR);
          break;
      case $KeywordRegistryCarPool.carColor$yellow():
          result.string(neon.PRIORITY_MEDIUM_COLOR);
          break;
      case $KeywordRegistryCarPool.carColor$red():
          result.string(neon.PRIORITY_HIGH_COLOR);
          break;
  }
  ```

- If you have not done it before, extend lib "KeywordRegistry_carPool" by the functions returning the KEYID of keyword entries corresponding to the colors:

  KeywordRegistry_carPool

  ```
  // Keyword entry names (KEYID)
  $KeywordRegistryCarPool.carColor$red = function(){return "RED";};
  $KeywordRegistryCarPool.carColor$yellow = function(){return "YELLOW";};
  $KeywordRegistryCarPool.carColor$green = function(){return "GREEN";};
  ```

- Now, you can extend the "Table" ViewTemplate of CarFilter_view by a further column holding the new EntityField COLOR_ICON. Deploy and see the result in the client.

11.6.4. Login web page

The client web page showing the ADITO login mask consists, by default, of

- a standard background, showing a marketing-style photo of ADITO,
- an ADITO logo, placed above the login mask, and
- the description "ADITO <Number>" in the browser tab title, e.g. "ADITO 2019".

If you want to customize these design elements, proceed as follows: In the "Projects" window of the ADITO Designer, navigate to system > default and, in the Editor window, double-click on "____CONFIGURATION". Then, in the Navigator window, navigate to System > Client. This will open a property list in the Editor window: Look at section "Client", which includes the following properties:

- clientTitleText: Here, enter the text you want to be displayed as title of the browser tab.
- clientLogo: If you want your own logo to be displayed above the login mask, choose it using the file browser, available via the three-dotted button.
- clientBackground: If you want your own background image to be displayed, choose it using the file browser, available via the three-dotted button.
- clientTimeout: Specifies the timeout of a client in milliseconds. If a user does not work within this time, the connection is disconnected and the client is terminated. The value is freely selectable as a positive integer.

11.7. Automatisms

Besides the design elements that can be controlled manually, ADITO includes various automatisms to ensure a well-balanced design.

11.7.1. Visibility of tabs

The visibility of a tab in a MainView is calculated automatically, in order to avoid empty tabs, as far as possible. The visibility of a tab depends on the visibility of the components in it: If there is a View reference, e.g., to a table and this reference is not visible, then the tab will also be invisible.

Example:
Person_entity: In the detail area of the MainView, there is a View reference to PersonTaskAppointment_view. This View, in turn, includes a View reference to the tasks (TaskFilter_view) and to the appointments (AppointmentFilter_view). If both tasks and appointments are invisible, then the tab is invisible. If at least one of these View references is visible, then the tab is also visible.

In any case, the reason for the invisiblity does not matter - be it because of permissions, Consumer state, settings of the "devices" property, or just because the View reference is empty.

The calculation of visibility also works for deeply nested Views - the principle remains the same: The visibility depends on the visibility of the View reference(s). The same applies to one-to-one Consumers (property isOneToOneRelationship set to true), as they are also realized as reference.

However, if a ViewTemplate of the original Entity is involved, its invisibility does not contribute to the (in)visibility of the View it is assigned to.
Example (fictitious):
Person_entity: In the detail area of the MainView, there is PersonAttributes_view (via View reference on #ENTITY). PersonAttributes_view, in turn, has 2 elements:

1. AttributeRelationFilter_view (via View reference on Attributes)
2. "Generic" ViewTemplate with 3 EntityFields of Person_entity, e.g., ROLE, LETTERSALUTATION, CONTACTTYPE

Now, if

1. AttributeRelationFilter_view is invisible
2. all 3 EntityFields of the "Generic" ViewTemplate are invisible (e.g., because of property "hideEmptyFields")
   then the tab nevertheless remains visible, but white and "empty".

To sum it up: The (in)visibility logic for tabs only works for View references loaded via a Consumer, but not for the direct usage of ViewTemplates. (This may change in future ADITO versions.)

Furthermore, there is a specific debug level available: NEON_COMPONENT_VISIBILITY

This debug level helps to analyze, why a component is visible or not: When dependencies are involved, there is a logging of changes of the visibility of View references and of the reason for these changes. The content of the logging will give you (in technical language) the following information: "Component PersonTaskAppointment_view is invisible, because TaskFilter_view is invisible and AppointmentFilter_view is invisible."

11.7.2. Visibility of drawers

The explanations in chapter Visibility of tabs are analogously valid for drawers: If a drawer has got no visible content (be it because of stateProcess, permissions, etc.), no empty drawer is shown.

12. 360Degree Context

The 360Degree_entity models the relations between specific Entities and enables the user to work with these dependencies via the 360DegreeFilter_view, which includes the ViewTemplates "Tree" and "Timeline".

Currently, the 360Degree logic is restricted to relations of Contact_entity (i.e., of companies and persons). This means that

- the 360Degree View can only be referenced in the MainViews of the Contexts "Organisation" and "Person" (appearing as tab "360 Degree");
- the 360Degree View can only include datasets (records) of Contexts having a relation to Context "Contact" (directly or via a "junction Context") - such as the Contexts "Salesproject", "Offer", or "Order" do.

In the xRM-Project, in property "documentation" of 360Degree_entity, you can find a comprehensive explanation of the basics and of how to extend the 360Degree View: Simply open its source text via the property’s three-dotted button and then, in the Editor (middle part of the Designer), choose tab "Preview". (It will take some seconds to open the preview.)

13. Internationalization

ADITO is perfectly suitable for being applied in an international context. Every textual element in its components, e.g., the title of an EntityField, can automatically be translated into the language defined in the user’s browser.

In addition to the following sub-chapters, we recommend you to read also the complete chapter "Internationalization" of the Designer Manual, where you will find additional information.

13.1. Language files

The core element of ADITO’s internationalization are translation files, separate for multiple languages, which are available in the folder "language" (see "Projects" window). These files contain key-value pairs, stored in XML format, in the editor window visible as table with 2 columns: "Key" and "Value":

- "Key" refers to the original text used in the designer, e.g., the value of the "title" property of an Entity. In xRM, all "Key" values are in English. We recommend to use English also for all terms (titles, placeholders, etc.) that are added or modified by customizing.
- "Value" refers to the text to be displayed in the client instead of the "Key".

You can easily add files for further languages by right-clicking on the "language" folder, choosing "New" in the context menu, and in the next step selecting "language" as type. Finally, select the required language from the language table, and press OK. Then, a new language file is created, which automatically contains all keys included in the existing files of other languages.

The usual approach is to let the keys be auto-generated, using the "Extract Keys" button (see below). Besides, you can create further keys manually by right-clicking into a language table and then choosing "Add row" from the context menu. New key-value rows first always appear on the top of the table, but after entering it, they are shifted according to the alphabetical order.

13.1.1. Refresh

If you click on the "Refresh" button (a button in the Editor window, showing a "refresh" symbol), all language files are scanned and compared, in order to assimilate their keys (or, in mathematical terms, build the "union set" of all keys of all language files). Then, all files will hold exactly the same keys, even if not all of them have translations (values).

13.1.2. Extract keys

If you click on the "Extract Keys" button (a button in the Editor window, showing a white "key" symbol), the whole ADITO system is scanned for textual elements that can be translated. New texts are added, but removed texts remain as keys.

The file "____LANGUAGE_EXTRA" is exclusively used for configuration purposes. Its keys and values are not used. But if you click on it and edit its property "sqlModels" (via the three-dotted button), a dialog titled "SQL" opens. Here, in the left part, mark "Data alias". Then, several SQL statements appear in the right part of the dialog. These statements will be executed when clicking on the "Extract Keys" button (see above), which results in the creation of additional "Keys", e.g., the titles of keyword entries.

13.1.3. Find unused keys

Click this button ("minus" icon) to scan all (!) language files for keys that are currently not used in the ADITO project. The respective keys are listed, and then you can decide which of them to delete.

13.1.4. Export/import

You can export and import keys in different char sets, with the option to specify what character to use as separator, etc.

13.1.5. Translate all

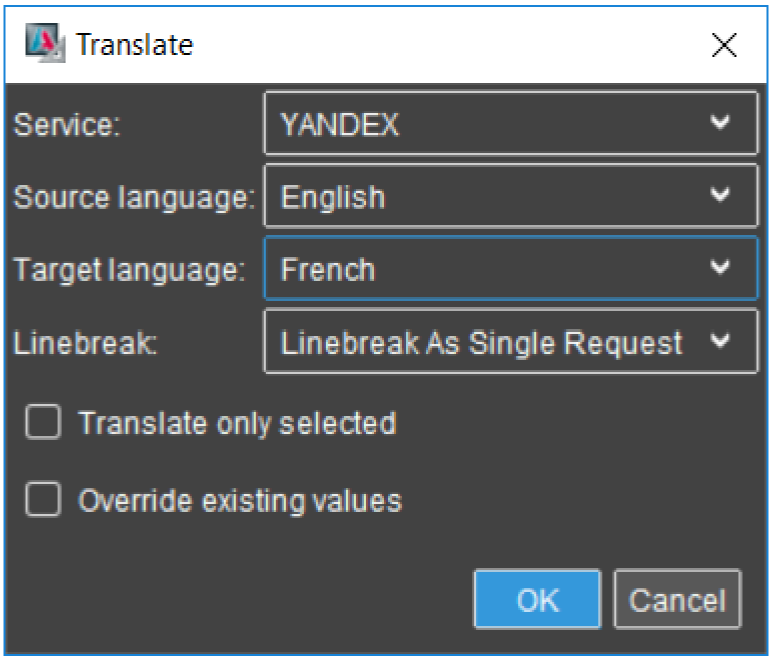
The button "Translate all" ("globe" icon) refers to only the language file you have currently opened. It enables you to let all keys of this file be translated automatically, i.e., to let ADITO retrieve and fill all "Values" automatically. This is done via a web service. Currently, ADITO supports 3 web services: Google, Yandex, and DeepL. To make them work, enter the respective API key under Tools > Settings > ADITO > Translators. (Find more information on how to obtain an API key on https://cloud.google.com/translate/, https://tech.yandex.com/translate/, and https://www.deepl.com/pro, respectively.)

After clicking the "Translate all" button, a "Translate" dialog appears (if you have specified an API key, see above), with the following parameters:

- Service: Select the web service to be used for the automatic translation.
- Source language: The language of the keys.
- Target language: The language of the values.
- Line break: Specify here, how ADITO will handle line breaks included in the keys.
    - Line break as single request (default): The text of a key will be cut into multiple parts, according to the line breaks. Each part will then be sent as separate request. This may increase the costs, if the maintainer of the translation web service charges you according to the number of requests.

    - Line break to space: Before the web service request, every line break will be replaced by a white space, i.e. line breaks will be lost in translation.
    - Disabled: No special handling for line breaks. This may result in the web service returning "strange" results or no result at all.

- Translate only selected: This option will automatically be checked, if you select one or multiple keys by marking them, then right-clicking on the selection and choosing "Translate…". In other cases, it has no effect.
- Override existing values: Check, if you want the web service result to overwrite the current content of the "Values".


Translate

Service: YANDEX

Source language: English

Target language: French

Linebreak: Linebreak As Single Request

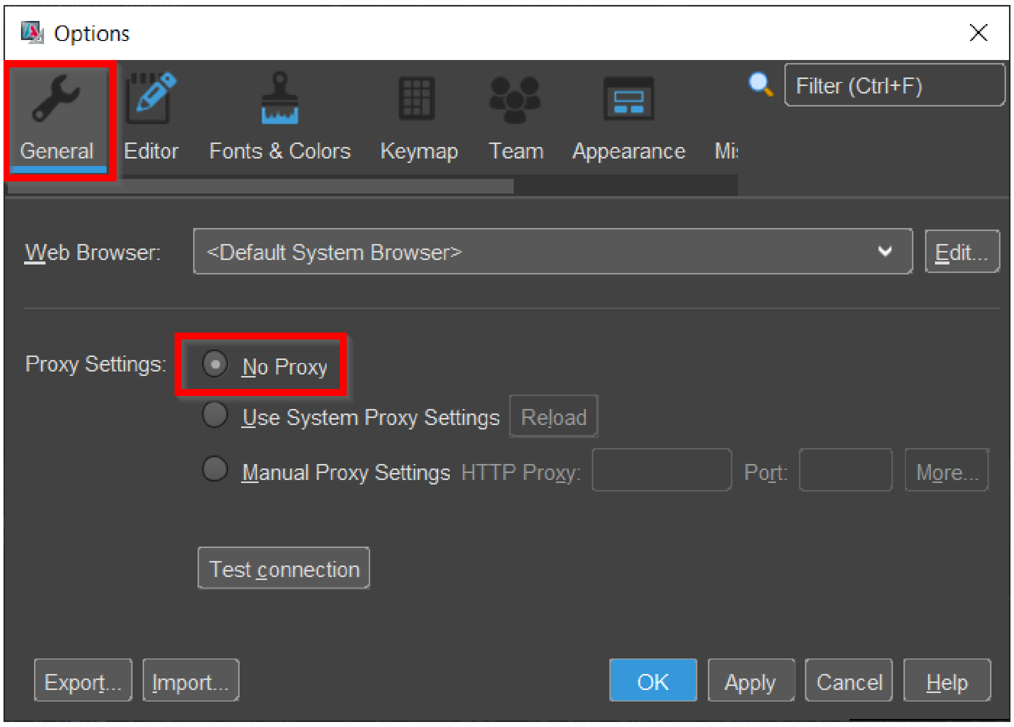
☐ Translate only selected

☐ Override existing values

Figure 37. Parameters of the automatic translation

If you want to use DeepL, please note:

- Usually, you do not need a proxy. On the basis of the key, ADITO automatically detects if the pro API or the free API must be called.
- Make sure that the web API is accessible, e.g., by executing the following test URL: https://api-free.deepl.com/v2/translate?auth_key=<API-Key>&text=HelloWorld&target_lang=DE (insert your API key accordingly before executing)
- Some IT environments require you to deactivate the proxy in the Designer options (Tools > Options > General > "No Proxy"):


Options

Web Browser: <Default System Browser>

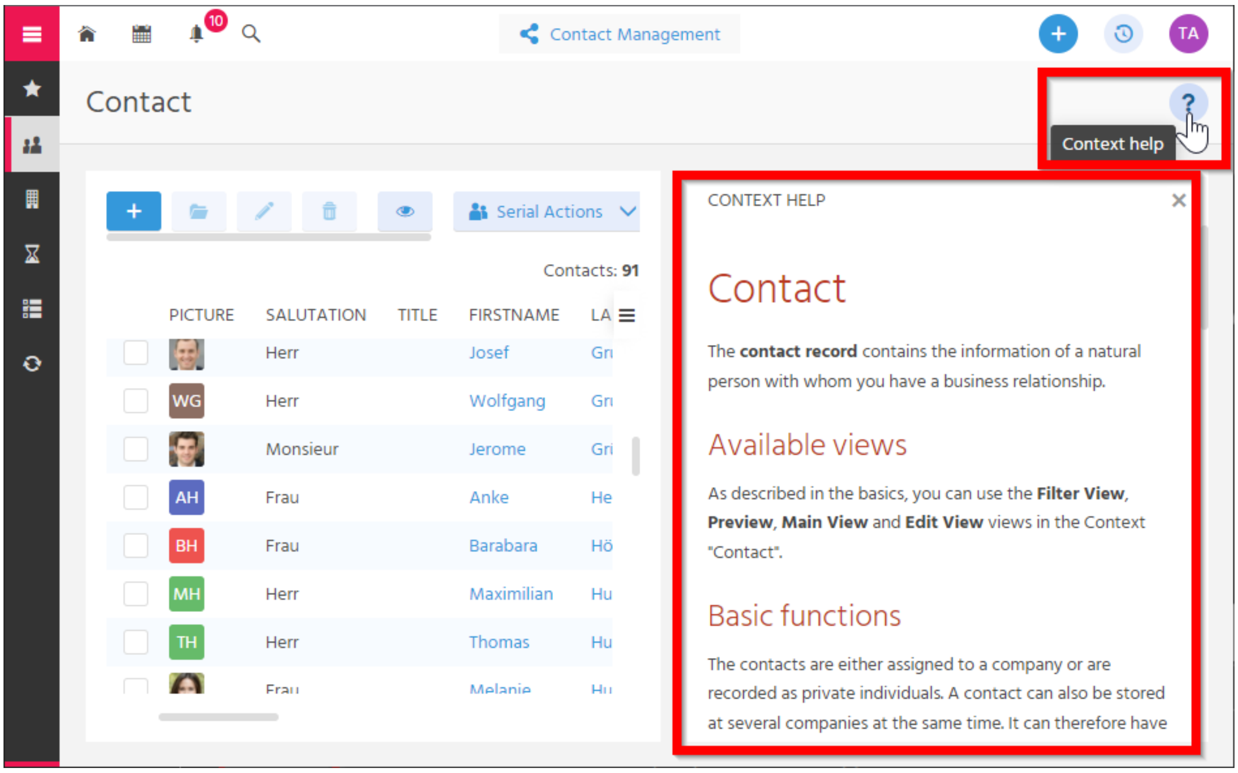
Proxy Settings: [✓ No Proxy] Use System Proxy Settings Manual Proxy Settings HTTP Proxy: Port:

13.2. User help

In the ADITO client, users can open help texts and illustrations, via the "questionmark" buttons. These are also subject to internationalization, i.e, they can be customized in order to be displayed according user’s browser language.

In ADITO document AID005_Userhelp.pdf, you can find more information of how to customize and maintain the "User help" via the ADITO Designer.

In the xRM project, this functionality is included, e.g., as "Context help":



PICTURE	SALUTATION	TITLE	FIRSTNAME	LA
	Herr		Josef	Gr
WG	Herr		Wolfgang	Gr
	Monsieur		Jerome	Gr
AH	Frau		Anke	He
BH	Frau		Barabara	Hö
MH	Herr		Maximilian	Hu
TH	Herr		Thomas	Hu
	Frau		Melanie	Hu

CONTEXT HELP

Contact

The **contact record** contains the information of a natural person with whom you have a business relationship.

Available views

As described in the basics, you can use the **Filter View**, **Preview**, **Main View** and **Edit View** views in the Context "Contact".

Basic functions

The contacts are either assigned to a company or are recorded as private individuals. A contact can also be stored at several companies at the same time. It can therefore have

Figure 38. Example of facilitating the "user help" functionality

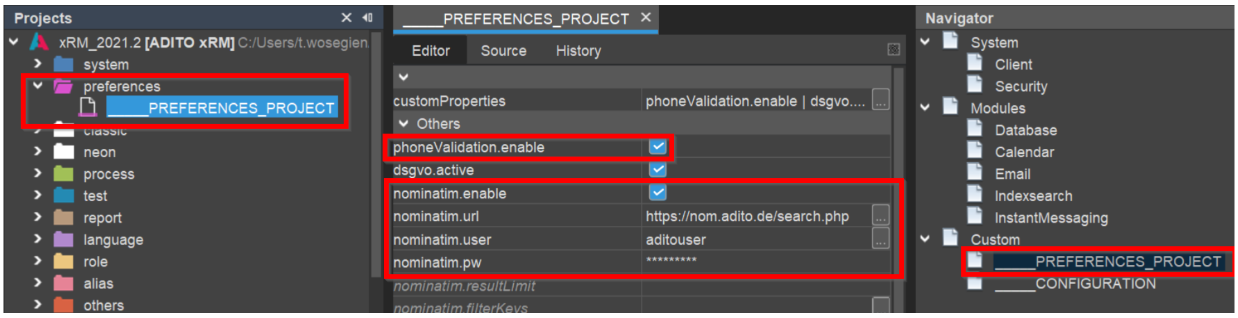
13.3. Validation of address and communication data

By default, the xRM project includes a country-specific validation of

- addresses
- communication data, like telephone number, email address, etc.

Example:
In PersonEditView, the ADITO logic will automatically check, if the address’s zip code and the telephone number comply with country-specific formats. (This functionality requires, of course, that the country has been set first, in the "standard" address.) If this validation fails, a message is shown beside the save button. And the save button will be disabled until the entered data has been corrected.

From the customizing point-of-view, there is nothing to do except for making sure that the respective validation properties are set correctly, under preferences > *__PREFERENCES_PROJECT > Custom >* __PREFERENCES_PROJECT:



customProperties	phoneValidation.enable | dsgvo....
Others	
phoneValidation.enable	✓
dsgvo.active	✓
nominatim.enable	✓
nominatim.url	https://nom.adito.de/search.php
nominatim.user	aditouser
nominatim.pw	*********
nominatim.resultLimit	
nominatim.filterKeys	

The actual validation methods (provided by the ADITO platform) are called in Communication_lib (under process > libraries). You may study this library for a deeper understanding of the topic.

As for the communication data, the following background information might be helpful:

All communication data (telephone number, mobile phone number, email address, website URL, etc.) are organized via the KeywordCategory "CommunicationMedium", which has the following KeywordAttributes:

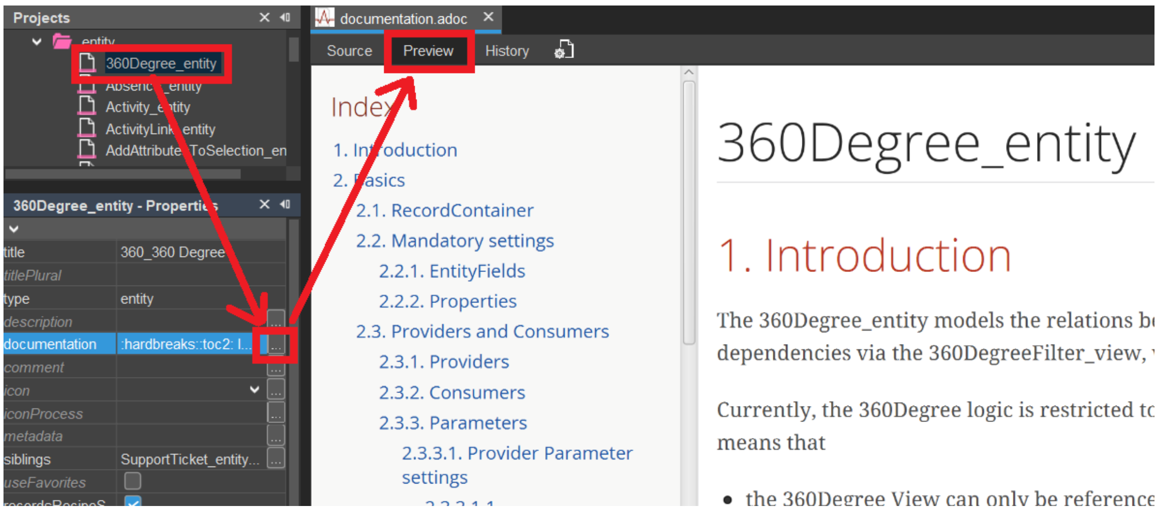
- **contentType** restricts the type of content that can be entered, and it determines its auto-formatting. There are the following contentTypes:
  - TELEPHONE: for telephone number, mobile phone number, etc. The automatic validation checks if the entered value complies with the country-specific format, and the country’s international area code will be automatically added at front.
  - EMAIL: for email addresses. The automatic validation checks if the entered value complies with the structure of a valid email format - e.g., if it includes an "@" and if its domain extension (".com", ".fr", ".de", etc.) exists. (But it will *not* check, if the username or the domain exists, or if the domain’s server is actually operating.)
  - LINK: for URLs of web sites, Blogs, Xing, LinkedIn, etc. The automatic validation checks if the entered value complies with the structure of a valid URL format (similar to the validation of email addresses, see above).
- **category**: PHONE, EMAIL, or OTHER. This enables a definition that only one single telephone number can be set as *standard* telephone number, and only one single email address can be set as *standard* email address. You can register, e.g., multiple telephone numbers (of private phone, company phone, mobile phone, etc.), but you can set only one of those as *standard*. (The consequences of setting this "standard" is explained in the client user-related documentation.)

- **placeholderTitle**: this value of the placeholder property of an EntityField, i.e., the text to be shown in the client as long as the user has nothing entered into the field.

14. Further information

Besides the documentation that is prerequisite for this manual (see chapter Prerequisites), you can find further information on ADITO customizing

- in the "documention" property of various ADITO components, e.g., 360Degree_entity;



entity
360Degree_entity
AbsenceEntity
Activity_entity
ActivityLink_entity
AddAttributeToSelection_en

title	360_360 Degree
titlePlural	
type	entity
description	
documentation	:hardbreaks::toc2: l...
comment	
icon	
iconProcess	
metadata	
siblings	SupportTicket_entity...
useFavorites	☐
recordsRecipeS	✓

Index

1. Introduction
2. Basics
2.1. RecordContainer
2.2. Mandatory settings
2.2.1. EntityFields
2.2.2. Properties
2.3. Providers and Consumers
2.3.1. Providers
2.3.2. Consumers
2.3.3. Parameters
2.3.3.1. Provider Parameter settings

360Degree_entity

1. Introduction

The 360Degree_entity models the relations b
dependencies via the 360DegreeFilter_view,

Currently, the 360Degree logic is restricted to
means that

- the 360Degree View can only be reference

- in the ADITO Information Documents (AID), which you can find in the customers' area of the ADITO website, see https://www.adito.de/login.html;
- in further documents, available via the respective overview article in the "Knowledge" area of the ADITO service client.

15. Troubleshooting

If you encounter problems, please make sure that you have

- taken part in the basic ADITO **training courses**, especially in the following ones:
  - client user
  - Designer
  - system operations
  - customizing
- **read the *latest* version of the Customizing Manual completely**:
  - The first part of this manual is designed like a schoolbook: On the basis of a plain example, you learn to handle ADITO step-by-step. **It is not recommended to skip one of these chapters**, as each chapter implies that you have read the previous ones.
  - The second part of this manual is more glossary-like: Additional knowledge is imparted using various best-practice examples included in the ADITO xRM project. Further helpful details are available in the appendices. Nevertheless, **we recommend you to read also these glossary chapters completely**, in order to have all required skills available.

Furthermore, you can find additional information in **topic-oriented documents**, e.g., in the Designer Manual, the Reporting Manual, the workflow documentation, client-user-specific documentation, as well as in the "ADITO Information Documents" (AIDs). ADITO will be happy to provide you with the latest version of these documents on request. (Most of them is also available in the customer area of ADITO’s web site, see https://www.adito.de/login.html .)

Last but not least, we recommend you to make sure that your ADITO contact has registered your email address for all relevant **newsletters**. In these, ADITO will inform you about new product features and new manual versions regularly.

In the following chapters, you will find further hints and tips for troubleshooting.

15.1. Built-in Designer help

The ADITO Designer provides you with the following built-in help functionality:

- Several **models** (Entities, Libraries, etc.) have explanations integrated in a property named "documentation" - some more, some less. A good example is the library "SqlBuilder_lib" in folder process > libraries. In its "documentation" property, you can find an extensive documentation of the SqlBuilder and its methods.
- In the property sheet, you can click on the name of a **property** and read a short text explaining

its purpose, right on the bottom of the "Properties" window.

- Both **methods** of xRM processes (e.g. `Utils.parseJSON(…)` of Util_lib and methods of the ADITO core (e.g., `entities.getRows(…)` of class system.entities) feature an JSDoc explaining the purpose and usage of the method, partly showing also an example code. You can access this documentation, if you type the method name in a code window, and then press CTRL+SPACE.
- Besides, **further descriptions** are given in specific parts of the Designer, e.g., in the "Add ViewTemplate" dialog.

> Whenever you encounter one of the above documentations to be unclear or not present where you had expected it, please write a note to your ADITO contact, who can then initiate an improvement of the documentation, to be released in a future version of the ADITO platform or the ADITO xRM project, respectively.

15.2. ScanServices

The ADITO Designer includes so-called ScanServices, which are processes running permanently in the background and scanning your ADITO project for code defects, inconsistencies, or other possible sources of errors. All results of the ScanServices are displayed in the window titled "Scan Services" in the lower middle part of the Designer (if not present, you can open this window by selecting it in the "Window" menu of the menu bar). A result line is marked with a yellow icon, if it is a warning, and with a red icon, if it is an error. You can "jump" to the source of the warning/error by double-clicking on the respective line, or by right-clicking on the line and then choosing "Open in Editor" in the context menu of the line. Mass-edit is also possible, if you multiselect result lines of the same type.

Via the buttons in the vertical button bar (left part of the "Scan Services" window), you can refresh or re-organize the structure of the result tree according to your own requirements.

An ADITO Entity including an error (e.g., in one of its properties) is underlined with a red wavy line in the "Projects" window.

If you open an ADITO project for the first time, a full scan starts automatically, which can take some minutes (you may notice this by high consumption of CPU resources). All further changes you perform in your project will then automatically be scanned immediately and individually. If you want to start another full scan manually, press the button with the "refresh" icon.

Find more information on ScanServices in the Designer Manual.

15.3. Bug tracking

When customizing your ADITO project, it cannot be completely avoided that you sometimes produce a bug. The central information for analyzing bugs is the so-called **stack trace**. In many cases, the error’s stack trace appears in the server log (be it visible in the Web Client or not). It looks like this (example):

Example stack trace of a bug

```
2022-04-25T08:44:36] [R-1-N-735-S] [<!--Entity: MySuperEntity_entity//-->] [<!--RecordContainer: db//-->] [<!--UID: null//-->] [<!--
Filter: {filter=null, filterCondition=null, ids=["e4be29ef-18d6-472d-aa27-050393c0f841"], excludedIds=null, permissions=null,
condition=MYTABLE.MYIDCOLUM IN ( 'e4be29ef-18d6-472d-aa27-050393c0f841' ) }//-->] [<!--Caused by:
de.adito.aditoweb.core.checkpoint.exception.AditoPermissionException: [R-1-N-735-S] [<!--Entity: ProjectticketComment_entity//-->]
[<!--RecordContainer: db//-->] [<!--UID: null//-->] [<!--Filter: {filter=null, filterCondition=null, ids=["e4be29ef-18d6-472d-aa27-
050393c0f841"], excludedIds=null, permissions=null, condition=MYTABLE.MYIDCOLUM IN ( 'e4be29ef-18d6-472d-aa27-050393c0f841' ) }//-->]
    at de.adito.aditoweb.server.neon.entity.EntityModel.init(EntityModel.java:430)
    at de.adito.aditoweb.server.neon.images.context.ContextEntityModel.init(ContextEntityModel.java:119)
    at de.adito.aditoweb.server.neon.images.common.AbstractEntityImageNeon.init(AbstractEntityImageNeon.java:64)
    at de.adito.aditoweb.server.neon.services.imagecontrol.command.open.AbstractImageOpenCommand.doOpen(AbstractImageOpenCommand.java
:99)
    at de.adito.aditoweb.server.neon.services.imagecontrol.command.open.AbstractImageOpenCommand.execute(AbstractImageOpenCommand.
java:55)
    at de.adito.aditoweb.server.neon.services.imagecontrol.command.open.AbstractImageOpenCommand.execute(AbstractImageOpenCommand.
java:19)
    at de.adito.aditoweb.server.neon.services.imagecontrol.ImageHandlingStrategy.handleImageOpened(ImageHandlingStrategy.java:73)
    at de.adito.aditoweb.server.neon.services.imagecontrol.ImageControlImpl.open(ImageControlImpl.java:62)
    at de.adito.aditoweb.server.neon.ClientSession.open(ClientSession.java:375)
    at de.adito.aditoweb.server.neon.entity.BaseEntityField._createPeviewWithUids(BaseEntityField.java:457)
    at de.adito.aditoweb.server.neon.entity.BaseEntityField._preview(BaseEntityField.java:400)
    ... 6 more//-->]
[C][B-54-N-112-S] [<!--de.adito.aditoweb.core.checkpoint.exception.mechanics.AditoException: [B-54-N-112-S]
    at de.adito.aditoweb.server.neon.entity.BaseEntityField._preview(BaseEntityField.java:407)
    at de.adito.aditoweb.binding.Action.call(Action.java:62)
    at de.adito.aditoweb.neon.base.vaadin.vclient.clientcomponents.images.frame.components.INeonComponent.lambda$doAction$3
(INeonComponent.java:214)
    at de.adito.aditoweb.neon.base.module.mcommon.UIWorker._loop(UIWorker.java:53)
    at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1128)
    at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:628)
    at java.base/java.lang.Thread.run(Thread.java:830)//-->]
```

As you can see, this stack trace consists of **a lot of** single log entries, which may overwhelm you at first sight. But don’t be discouraged! You will know learn how to analyze it, in order to detect the source of the problem. Consider yourself to be a "tracking dog", "sniffing" along the (stack) trace until you find the target. If you know what to sniff for, you reach your goal precisely and successfully. Rest assured: After gaining some experience, you will soon be able to detect the source of an exception quickly.

Now, typically, a stack trace includes…

- …one log entry naming the method or function in which the actual exception happened. This is what you have to look ("sniff") for. However, this line is not always easy to find, because it is surrounded by…
- …a lot of further log entries, which include functions or methods that were called before or after the actual exception. Hence the name "stack **trace**".

To find you way through this "woods" of log entries, an approach in the following order has turned out to be pragmatic:

- Make sure you are familiar with all functions of the ADITO Designer’s build-in debugger (see the ADITO Designer Manual). Trying to track errors without knowing how to use the debugger is like being a tracking dog without a nose.
- If you can reproduce the error, first clear the server log: Right-click in the server’s log window and choose option "Clear" (NOT "Clear cache"!), in order to delete all earlier log entries. If you then reproduce the error, you will see only the log entries that are really related to the error.
- Stack traces are often logged repeatedly. Therefore, you only need to concentrate on the first set of log entries, ignoring their repetitions.
- Roughly scan through the stack trace and see if you find somewhere any sentence explaining the error in human language.
  Example: (…) `SqlBuilder: .where has to be called before following and/or.` (…) Then you know that the problem is and can fix it, ignoring the rest of the stack trace.
- Scan the stack trace for specific exception names that start with "Adito". If you, like in the above example, read "AditoPermissionException", then at least you know that your problem is related to a conflict with permissions (= access rights, configured via the Client’s menu group "User Administration").
- Concentrate on the log entries that do NOT start with the word "at". These often include automatically generated information, like shown in the above example: In this example, you see that the problem
  - is related to a specific Entity (here: "MySuperEntity_entity")
  - occurs when EntityRecordsRecipe is used (which you can conclude from log elements like `ids`, `excludedIds`, and `filterCondition`)
  - Now you know that you can restrict your bug tracing to methods that you have newly added and that use EntityRecordsRecipe (as parameter), e.g., method `neon.openContextWithRecipe` - even if this method is not excplicitely given in the stack trace.
  - In this example, the next step would be to activate the debugger and stop at these very methods, e.g., in order to analyze their parameter values.
- Concentrate on log entries that start with `de.adito`…. Ignoring **all** log entries that start with something else, especially `java`… (These are related to the Java engine and only show the consquences of the error, not its actual source.)
  - Scan these remaining log entries for names of models (Entities, Actions, etc.), processes, methods, or properties that are part of your project - especially those that you have lately

created, used, filled, or changed.

  - If this search is successful, check exactly the code line whose number is given at the end of the log entry. Example: If you read in the log entry `Activity_entity.entityFields.testAction.onActionProcess#13)` then you know that the error occurs at line 13 of property "onActionProcess" of Action "testAction" of Entity "Activity_entity"
  - If you inspect this code line, you will possibly detect the source of the error immediately, because
    - the error is obvious and/or
    - this code line is marked with a red wavy line, and if you hover over this line with your mouse pointer, a tooltip will pop up giving hints on the error’s source
    - a yellow light bulb is shown to the left of the code line: If you hover over it with the mouse pointer, a tooltip will pop up giving hints on the error’s source. And if you click on the light bulb, possible solutions for the problem are offered, which you can click on, if this solution seems right to you.
  - If you still cannot find the error, then activitate the debugger, place a halting point at the respective code line, and try to reproduce the error. As soon as the halting point is reached, inspect the variable values / parameter values at this state of excecution. This will help you to identify the problem.

- If the stack trace suggests that the source of the problem is not related to a process, function, or property of your project, but to a method of the ADITO platform (often called "core method"), then first check, if you have called this method with valid parameters (using the Debugger, see above). If this is true, then report the problem to ADITO by issuing a bug ticket via the ADITO Service Client).
- Generally, you should not restrict your inspection to the code line causing the error, but also find out on which way this line was reached: If, e.g., the error happens in an certain function of an xRM library, it might be helpful to know what user input (e.g., an executed Action) called this function. For example, the user might have entered "0", and this input is passed as parameter to a function, where "0" causes an error - then it is not enough to `catch` "0" in the function, but you should also validate the user input in the client and prevent that "0" is entered, along with a suitable validation message. Or, if the user leaves an input field empty, which might cause a NullPointerException, the solution should include to mark the corresponding EntityField as mandatory.
- If you have fixed the problem, check if the logging could be improved in order to find and fix similar errors easier. Find more information in chapter Logging. If necessary, help ADITO to improve the product by issuing a ticket via the ADITO Service Client.

Further examples:

Enter the following faulty code in the onActionProcess of any test Action

Faulty example code

```
var myVariable1 = "Test";
myVariable1 = null;

// here, an exception will occur
var myVariable2 = myVariable1.toString();

question.showMessage(myVariable2);
```

If you excecute this Action, you might be shocked to see the following long stack trace both in the client and in the Designer’s server log window:

Stack trace produced by faulty code

```
Z-00-N-0011-S Original java exception causing the error. TypeError:
Cannot call method "toString" of null (Activity_entity.entityFields
.testAction.onActionProcess#8) [ID ee2a0441-2d82-3d55-8ad1-
eb47888082e5]  [->] Caused by: org.mozilla.javascript.EcmaError:
TypeError: Cannot call method "toString" of null (Activity_entity
.entityFields.testAction.onActionProcess#8)
    at org.mozilla.javascript.ScriptRuntime.constructError
(ScriptRuntime.java:4280)
    at org.mozilla.javascript.ScriptRuntime.constructError
(ScriptRuntime.java:4258)
    at org.mozilla.javascript.ScriptRuntime.typeError(ScriptRuntime
.java:4291)
    at org.mozilla.javascript.ScriptRuntime.typeError2(
ScriptRuntime.java:4310)
    at org.mozilla.javascript.ScriptRuntime.undefCallError
(ScriptRuntime.java:4327)
    at org.mozilla.javascript.ScriptRuntime
.getPropFunctionAndThisHelper(ScriptRuntime.java:2573)
    at org.mozilla.javascript.ScriptRuntime.getPropFunctionAndThis
(ScriptRuntime.java:2566)
    at org.mozilla.javascript.Interpreter.interpretLoop(Interpreter
.java:1537)
    at org.mozilla.javascript.Interpreter.interpret(Interpreter.
java:1013)
    at org.mozilla.javascript.InterpretedFunction.call
(InterpretedFunction.java:109)
    at org.mozilla.javascript.ContextFactory.doTopCall
(ContextFactory.java:412)
    at org.mozilla.javascript.ScriptRuntime.doTopCall(ScriptRuntime
.java:3578)
    at org.mozilla.javascript.InterpretedFunction.exec
(InterpretedFunction.java:121)
    at de.adito.aditoweb.jdito.interpreter.jscript.jsscript
.ScriptManagerNext.executeScript(ScriptManagerNext.java:51)
    at de.adito.aditoweb.jdito.interpreter.jscript
.AbstractJScriptInterpreter._evaluate(AbstractJScriptInterpreter.jav
a:298)
    at de.adito.aditoweb.jdito.interpreter.jscript
.AbstractJScriptInterpreter._evaluate(AbstractJScriptInterpreter.jav
a:240)
    ... 13 more
   J-03-D-0756-S JDito-error:
Unable to interpret code.  [->] Caused by: de.adito.aditoweb.jdito
.AditoJDitoException: [J-3-D-756-S]
    at de.adito.aditoweb.jdito.interpreter.jscript
.AbstractJScriptInterpreter._evaluate(AbstractJScriptInterpreter.jav
a:263)
    at de.adito.aditoweb.jdito.interpreter.jscript
.AbstractJScriptInterpreter._interpret(AbstractJScriptInterpreter.ja
va:175)
    at de.adito.aditoweb.jdito.interpreter.jscript
.AbstractJScriptInterpreter.interpret(AbstractJScriptInterpreter.jav
a:119)
    ... 11 more
      J-03-D-0040-S JDito-error:
Unable to interpret code.  [->] Script: Activity_entity.
entityFields.testAction.onActionProcess  [->] Caused by: de.adito
.aditoweb.jdito.AditoJDitoException: [J-3-D-40-S] [<!--Script:
Activity_entity.entityFields.testAction.onActionProcess//-->]
    at de.adito.aditoweb.jdito.interpreter.jscript
.AbstractJScriptInterpreter.interpret(AbstractJScriptInterpreter.jav
a:127)
    at de.adito.aditoweb.jdito.JDito.interpret(JDito.java:53)
    ... 10 more
         R-03-R-0008-S JDito-error:
Unable to interpret code.  [->] ResultType: 5  [->] Caused by: de
.adito.aditoweb.jdito.AditoJDitoException: [R-3-R-8-S] [<!--
ResultType: 5//-->]
    at de.adito.aditoweb.jdito.JDito.interpret(JDito.java:59)
    at de.adito.aditoweb.server.neon.entity.calculation
.JDitoFunction.call(JDitoFunction.java:45)
    at de.adito.aditoweb.server.neon.entity.BaseAttributeField
.lambda$registerActionProcess$0(BaseAttributeField.java:235)
    ... 8 more
            P-54-R-0004-S Error executing action.  [->] de.adito
.aditoweb.core.checkpoint.exception.mechanics.AditoException: [P-54-
R-4-S]
    at de.adito.aditoweb.core.checkpoint.CheckPointHandler
```

```
.checkPoint(CheckPointHandler.java:114)
    at de.adito.aditoweb.server.neon.entity.BaseAttributeField
.lambda$registerActionProcess$0(BaseAttributeField.java:242)
    at de.adito.aditoweb.binding.IActionCallable.call
(IActionCallable.java:26)
    at de.adito.aditoweb.binding.IActionCallable.call
(IActionCallable.java:16)
    at de.adito.aditoweb.binding.Action.call(Action.java:44)
    at de.adito.aditoweb.neon.base.vaadin.vclient.clientcomponents
.images.frame.components.buttonstrip.NeonButtonStripUtil.lambda$exec
uteAction$0(NeonButtonStripUtil.java:153)
    at de.adito.aditoweb.neon.base.module.mcommon.UIWorker._loop
(UIWorker.java:53)
    at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker
(ThreadPoolExecutor.java:1128)
    at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run
(ThreadPoolExecutor.java:628)
    at java.base/java.lang.Thread.run(Thread.java:830)
```

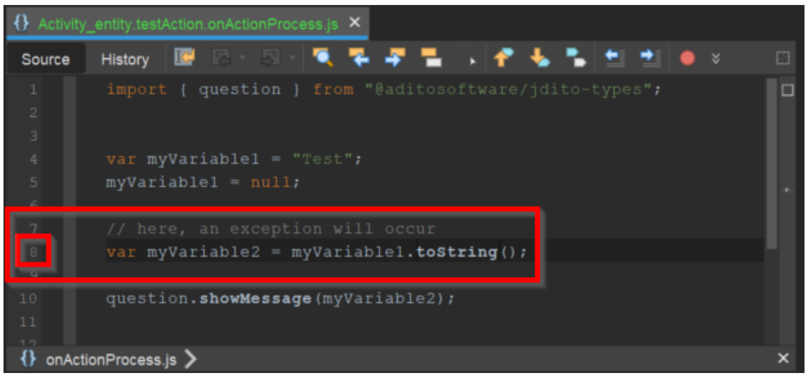
If you proceed as advised above, you will be able to extract the only important log entry (ignoring all the other stuff):

```
Cannot call method "toString" of null
(Activity_entity.entityFields.testAction.onActionProcess#8)
```

In clear text, this log entry means:

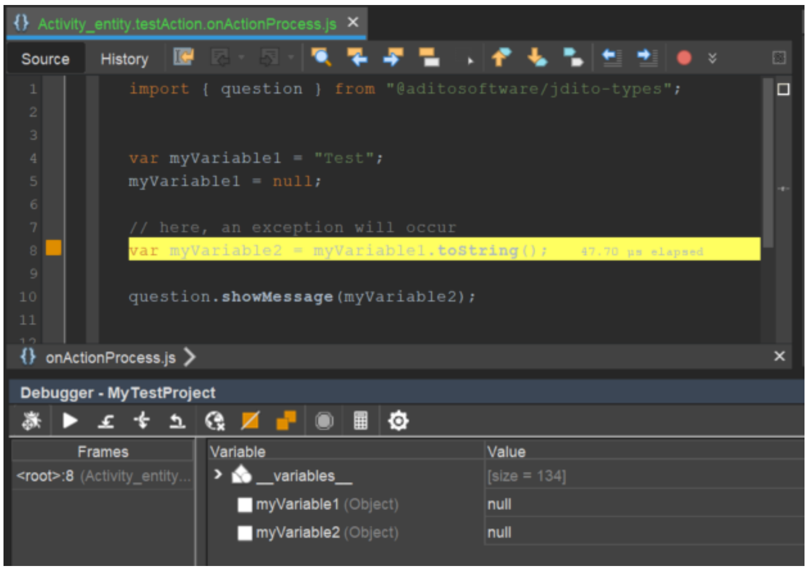
- The error happens at line 8 of property "onActionProcess" of Action "testAction" of Entity "Activity_entity".
- The problem is, that method "toString" is called on a variable or an object that has the value null.

If you then navigate to this onActionProcess, you will easily be able to identify the source of the error:



```
1   import { question } from "@aditosoftware/jdito-types";
2
3
4   var myVariable1 = "Test";
5   myVariable1 = null;
6
7   // here, an exception will occur
8   var myVariable2 = myVariable1.toString();
9
10  question.showMessage(myVariable2);
11
```

Additionaly, you can halt at line 8 of this process, using the debugger, and inspect the variable values:



```
import { question } from "@aditosoftware/jdito-types";


var myVariable1 = "Test";
myVariable1 = null;

// here, an exception will occur
var myVariable2 = myVariable1.toString();    47.70 µs elapsed

question.showMessage(myVariable2);
```

Frames
<root>:8 (Activity_entity...

Variable	Value
__variables__	[size = 134]
myVariable1 (Object)	null
myVariable2 (Object)	null

15.4. Specific problems

15.4.1. Low performance

If your ADITO system is performing poorly, this could have various reasons. For example, if it takes long time to load the content of a table, the reason might be that

- the table includes an EntityField whose value is calculated via a complex valueProcess (instead of using the "expression" property of the RecordFieldMapping in the RecordContainer);
- the related database table needs indices to be set;
- the ADITO server has not enough main memory.

> We strongly recommend you to read the ADITO Information Document AID066 Performance Optimization. In this document you will find hints and advice on how to optimize the performance of your ADITO system.

15.4.2. Changes are not visible in the client

If you have modified your application in the Designer, but the changes are not visible in the client, try the following steps in the given order:

- Make sure that you have saved *and* deployed all changes, using button "Deploy Project" in the Designer’s button bar.
- Always re-open the web page on which you expect your changes to be visible, by clicking on the respective menu entry. (It is *not* enough to use the "refresh" button of your browser!)
- Log out and log into the client and open the respective page again.
- (In very rare cases:) Restart the ADITO server, proceeding as follows:
  - Choose "Server - default" from the combo box in the button bar and press the green triangle to the right of it. A dialog will appear asking you if the currently running server instance should be stopped, which you need to confirm. In the "Output" window (lower right part of the Designer), a new sub-windows "Server" will open, where you can watch the new server instance starting, until the log shows "Server initialized".
  - Alternatively to restarting in one step, you can restart the server in subsequent single steps:
    - Open the "Output" window (lower right part of the Designer)
    - Open the sub-window "Server".
    - Click on button "Exit" (white square icon), view the server log entries, and wait until you see the log "Server terminated". If this does not appear after a few

seconds, click button "Stop" (white cross icon).

    - Choose "Server - default" from the combo box in the button bar and press the green triangle to the right of it, view the server log entries, and wait until you see the log "Server initialized". (Please note: Clicking on button "Start" (white triangle icon in window "Output - Server", might not be enough, because in this case, less properties are being reloaded.)

  - Re-open the client: Select "Web Client (Neon)" in the combo box of the Designer’s button bar, then click button "Execute…" (green triangle icon). This will re-start the client, in which you can open the respective web page.

- In rare cases, and depending on the browser you use, it can also be necessary to empty the browser’s cache - in Google chrome, e.g., via the shortcuts CTRL+F5 ("deep refresh" of the current web page) or CTRL+SHIFT+DEL > "Clear data" (deleting selected cache data) - in order to see the changes effected by the deploy.

- Check if all configurations visible in the designer are consistent to the XML source code (this is the format in which all ADITO configurations are actually stored).
  Example:
  If you have entered the title of an Entity (in the "Properties" window), saved and deployed it, but you do not see it in the client, then

  - double-click on the Entity (in the "Projects" window)

  - open the tab "Source" in the Editor window (middle part of the designer)

  - search for the tag `<title>` and check if it includes the specified title. If not, you can either write it in the XML manually, or - better in most cases - force ADITO to re-synchronize Designer display and XML source code:

    - Close the Designer.

    - Navigate to the Roaming folder ".aditodesigner", open the folder corresponding to your ADITO version number, and delete its sub-folder "cache". (By default, this folder resides in the AppData\Roaming folder of the Windows user directory, e.g. C:\Users\j.smith\AppData\Roaming.)

    - Re-start the Designer.

    - Re-enter the title. (Usually, this should be necessary, because the re-synchronisation updates the display according to the XML source, not vice versa).

    - Re-check the XML source code, if the title is now present in the tag `<title>`.

    - If the problem persists, repeat the last steps, but this time, delete the complete folder corresponding to your ADITO version number (sub-folder of folder ".aditodesigner").

- If the problem still persists, repeat the last steps, but this time, delete the complete ".aditodesigner" folder.

From ADITO version 2022.1.0, folder ".aditodesigner" has one separate sub-folder (sub-user-directory) for every installed ADITO version, be it a major release (e.g., 2022.0), a minor release (e.g., 2022.0.1), or a hotfix (e.g., 2022.0.0.2). Earlier versions had sub-folders (sub-user-directories) only for every major release.

15.4.3. New database structure is not accessible

If you have changed the database structure via DML command, while the ADITO server is running (e.g., you have added a new database column via `alter table MYTABLE add column NEWCOLUMN`), then you must first delete the corresponding cache before you can access the new structure’s elements (e.g., the new column). Otherwise, you will get an error message stating that the respective structure element is not found, if you try to access it.

To delete the cache, open the ADITO Manager (in the client’s Global Menu, choose "Server" in menu group "Manager"), mark your server and choose option "Clear cache" via the three-dotted button in the PreviewView. Alternatively, of course, you can achieve the same effect by re-starting the server.

This is required, because, for performance reasons, the ADITO server does not permanently update the actual database structure.

Appendix A: JDito system modules and variables

A.1. System modules

Everything besides the basic functionality of JavaScript is provided by JDitos system modules. The methods are grouped by topic. For example, system.calendar contains every method used for interfacing with the calendar, system.db contains every method to interface with databases.

In general terms, a module contains every method and constant associated with its topic.

Constants are a kind of wrapper for badly readable values and are used so anyone who reads the code knows what the meaning of a specific value is. For example: The database type for Apache Derby is the numeric value "7", but there is a constant in the system.db module called "db.DBTYPE_DERBY10", which represents the same value, but has a much better readability.

Methods are used to tell the ADITO system what to do, like collecting data from a database or changing the value of a component. They’re not to be confused with functions. Functions are written in JDito and describe an order of commands, while JDito methods are implemented into the interpreter to be able to control the ADITO system.

To use a specific module, you need to import it into your JDito process using the `import` command.

The import commands must **always** be at the very top of a process, otherwise an error is thrown.

Example:

```
import { result } from "@aditosoftware/jdito-types";

//code begins here

result.string("My result");
```

The different system modules are:

System module	Description
system.ACTION	This module only contains constants that hold values for different client actions. Examples: ACTION.FRAME_CREATE, ACTION.FRAME_EDIT

System module	Description
system.ALIAS	This module only contains constants that are used for referencing different keys within an alias object. Examples: ALIAS.CHARSET, ALIAS.PASSWORD
system.SQLTYPES	In this module are constants for the different SQL columntypes and methods that help in validating the type, like SQLTYPES.isTextType(SQLTYPES.CHAR).
system.calendars	This module groups methods and constants necessary for working with calendars and calendar entries.
system.cti	In this module you can find all methods and constants needed for working with any telephone system.
system.datetime	Datetime holds methods and constants for working with dates, timestamps and time zones.
system.db	This module contains constants and methods for communication with connected databases, like executing selects, updates, and deletes.
system.eMath	eMath contains constants for different rounding behaviours and methods, that can savely add, subtract, multiply, divide and round data of the data type String. Each type of calculation exists seperately for integer and decimal values.
system.fileIO	Here you can find constants and methods for **serverside** file input / output operations.
system.gantt	This module currently has no contents.

System module	Description
system.im	In this module you can find methods and constants regarding the XMPP Backend.
system.imClient	In this module you can find methods and constants regarding the XMPP Client.
system.indexsearch	In here are the methods and constants used for running the indexing processes and for executing searches using the Solr indices.
system.logging	This module holds all methods and constants regarding the logging system.
system.mail	This module contains methods and constants for sending emails and working the email objects.
system.neon	This module contains methods and constants to control the neon web client.
system.neonTools	This module contains methods to assist in controlling the neon client.
system.net	net contains methods used for calling web services using REST or SOAP, for validating URLs and for getting the contents of a URL.
system.notification	Here you can find all methods and constants for controlling the notification system.
system.pack	pack offers methods and constants for working with ZIP archives.
system.plugin	This module contains methods for calling ADITO plugins on the **serverside**.

System module	Description
system.process	In this module you can find all methods and constants regarding the immediate execution of "executable" JDito processes and for managing the timed execution.
system.project	This module offers methods and constants for getting data models from the deployed project, like `getAlias()`, `getDatamodel()`, `getInstanceConfigValue()`.
system.question	This module holds the methods and constants for displaying different modal dialogs.
system.report	In this module are all methods and constants to interface with the reporting engine.
system.result	This module holds the methods used for returning values to the ADITO core. Mainly used in processes like valueProcess or displayValueProcess.
system.text	text contains methods for decoding and encoding multistrings, formatting and parsing text and for hashing.
system.tools	Despite its name, there are no "tools" in this module. Instead it holds all methods and constants used for managing users and roles.
system.translate	This module is used for calling the translation system. It contains all necessary methods and constants.
system.treetable	This module currently has no contents.

System module	Description
system.util	This module contains utilitary methods and constants. Mostly used for encoding and decoding BASE64 Strings and generation of IDs with getNewUUID().
system.vars	This module offers methods for reading and setting values from components, EntityFields, and system/image/local variables.

A.2. System variables

System variables in general (these are not only variables named "$sys.xxx") are containers for values that are either provided by the ADITO application core or are set via JDito Code.
They are read by method `vars.get()` and set by method `vars.set()` of the system module `vars`. Their name always has to be prefixed by the "$" sign.

Example:

```
// Reading a variable
vars.get("$sys.operatingstate");
vars.get("$field.UUID");

// Setting a variable
vars.set("$context.calculatedVal", calcValue);
```

`vars.getString()` is no longer required, even for reading String values. This method will be marked as deprecated in future ADITO versions.

There are four types of system variables ($image and $comp are not mentioned here, because they belong to the ADITO Legacy platform):

1. $local
   These system variables' session runtime is as long as the user is logged-in (per login). The runtime of the server environment depends on the configuration of the server process (must always be executed using a specific user) - e.g., in the process `autostartNeon`, various variables are being set that are required in the client.

   Mostly, $local variables are set by the ADITO core to pass values into specific processes. For example: process_audit is the process where you can react to the auditing of a specific dataset.

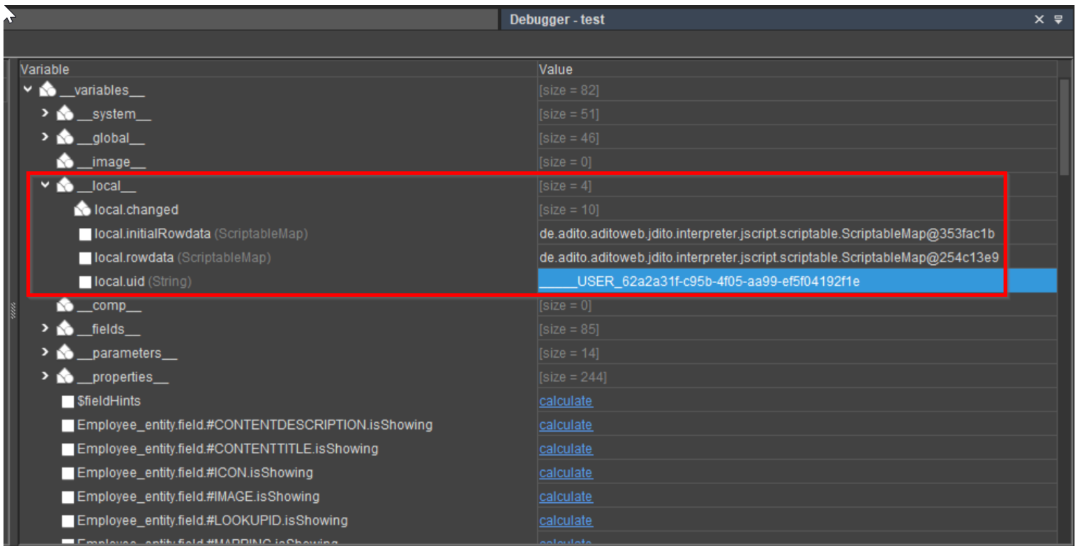
This process gets the information what database columns are affected, what their types are and what their old and new values are. These values are passed as local variables and are accessed via the system.vars module. Example:

```
//getting the value
var action = vars.get("$local.action");
//setting a value
vars.set("$local.taskId", "1234-1234-1234");
```

As for the server, this means:
When you configure the interval of a server process, you can specify, if the "JDito instance" should be kept:

- If yes, then the global variable persists over multiple runs (on the server - with "server" meaning one single server pod, e.g. `adito-web-bg-0`, i.e., the software-sided instance of an ADITO server);
- if no, then the global variable does not exist anymore for every new run and therfore must be set anew.

  It would be an extensive and hardly-to-maintain documentation to give a complete summary of all $local variables, along with their availability and purpose in each single context. Therefore, you will find only selected $local variables in appendix $local variables - but beside this, we recommend the "trick" to simply use the Designer’s debugger: In window "Debugger" (visible only if the debugger is active) you can inspect all $local variables that are available in the context of the respective breakpoint, where the debugger has halted. Here is an example:



Debugger - test

Variable	Value
__variables__	[size = 82]
__system__	[size = 51]
__global__	[size = 46]
__image__	[size = 0]
__local__	[size = 4]
local.changed	[size = 10]
local.initialRowdata (ScriptableMap)	de.adito.aditoweb.jdito.interpreter.jscript.scriptable.ScriptableMap@353fac1b
local.rowdata (ScriptableMap)	de.adito.aditoweb.jdito.interpreter.jscript.scriptable.ScriptableMap@254c13e9
local.uid (String)	_____USER_62a2a31f-c95b-4f05-aa99-ef5f04192f1e
__comp__	[size = 0]
__fields__	[size = 85]
__parameters__	[size = 14]
__properties__	[size = 244]
$fieldHints	calculate
Employee_entity.field.#CONTENTDESCRIPTION.isShowing	calculate
Employee_entity.field.#CONTENTTITLE.isShowing	calculate
Employee_entity.field.#ICON.isShowing	calculate
Employee_entity.field.#IMAGE.isShowing	calculate
Employee_entity.field.#LOOKUPID.isShowing	calculate

  (In the Designer Manual, you can find more information on the debugger’s functionality and handling.)

2. $sys
   The $sys variables are visible within one client and are independent from a specific context. They are typically used to store values that are used throughout the client, like global configurations, rights management via sales areas, etc.
   $sys variables are accessed as follows:

```
//getting the value
var action = vars.get("$sys.useRights");
//setting a value (only works, if SALESAREA is a dynamic user
property!)
vars.set("$sys.salesArea", tools.getCurrentUser()[tools.
PARAMS]["SALESAREA"]);
```

   In appendix "$sys variables", you can find a summary of all $sys variables, along with their individual purpose.

3. $field
   Field variables are the equivalent to $comp variables in Legacy ADITO, but they are used within the Neon Entity environment.

   An EntityField value can only be read. For setting the value of an EntityField refer to the corresponding method in the system.neon module.

   Example:

```
//getting a value
var id = vars.get("$field.UUID");
//setting would lead to an error!
```

4. $context
   These are variables that are valid during the runtime of a Context (neon data model) - i.e., e.g., what you have opened in a MainView.

5. $cluster
   Cluster variables are visible within **all** clients. They are set by the server. These are typically used as caches or for values that have to be the same on all clients. They are accessed in the same way as sys variables.

   Example:

```
//getting a value
var supportEmail = vars.get("$cluster.supportMail");
//setting a value
vars.set("$cluster.supportMail", "support@adito.de");
```

6. $image

   Deprecated. These variables were only used in the legacy client (equivalent to $contect).

> Variables can always be set only in the context in which they can also be read.
> It is not possible to set from one context (e.g., "Organisation opened in tab 1") the variable of another context (e.g., "Organisation opened in tab 2").
> Therefore, the possibilities to use globale variables as cache are very limited, because the invalidation is difficult - usually, this happens after a re-login in the `autostartNeon` process (or by a respective ServiceImplementation for this process).

Appendix B: Database Access

Please remind that, for every database table, an appropriate setting of **database indices** is required, in order to ensure an optimal performance of database access. Find further information about performance optimization in AID066.

B.1. Basic SQL Statement

Although SQL is used in many parts of the ADITO application, there is one basic SQL statement that is executed when a Context is opened, e.g., to display data in the FilterView. This basic SQL statement has got, as most SQL statements, the following clauses (parts):

- a SELECT clause (including all columns to be load)
- a FROM clause (including all involved tables)
- an optional WHERE clause (including one or multiple conditions)
- an optional ORDER BY clause (including one or multiple columns to use as order criteria)

```sql
SELECT MYTABLE.MYCOLUMN1, MYTTABLE.MYCOLUMN2, (...)
FROM MYTTABLE
WHERE MYCOLUMN1 = 'myParameter1' AND MYCOLUMN2 = 'myParameter2'
ORDER BY MYTTABLE.MYCOLUMN2
```

In ADITO,

- the SELECT clause is defined in the RecordContainer’s RecordFieldMappings MYFIELD.value and MYFIELD.displayValue, with the properties
    - recordfield: one specific database column to load
    - expression: an SQL expression to use instead of a specific column (marked with "(…)" in the above example SELECT). If the `result.string()` argument of this property’s code is simply `"MYTABLE.MYCOLUMN1"`, then the effect is the same as if we had selected MYTABLE.MYCOLUMN1 as recordfield. However, you can also enter advanced SQL code here, e.g.
        - to concatenate multiple columns, e.g. `result.string("MYTABLE.MYCOLUMN1 || MYTABLE.MYCOLUMN1")`
        - to enter a sub-select, e.g., `SELECT … FROM … WHERE`… (Caution: Depending on the kind of sub-select, this will be executed for every single dataset, which may decrease the performance).

- the FROM clause is specified in the following properties of the RecordContainer:
    - linkInformation: one or multiple tables to which the specified columns (see above) belong to. **Caution**: If you specify more than one table here, the cross product of all tables is loaded by default, which can result in huge data masses and therefore be a performance killer; therefore, multiple tables should only be specified along with additional properties, especially fromClause (with, e.g., JOINs) and conditionProcess (see below).
    - fromClauseProcess: Optionally, you can enter the complete FROM clause here (without the word "FROM" itself). JOINs may be included. All involved tables must nevertheless be specified in the property linkinformation. As usual for processes, the SQL must be specified as argument of method `result.string()`, e.g., `result.string("MYTABLE JOIN OTHERTABLE ON (…)")`.
- the WHERE clause can optionally be specified in the property conditionProcess (without the word "WHERE" itself). As usual for processes, the SQL must be specified as argument of method `result.string()`, e.g., `result.string("MYTABLE.MYCOLUMN1 = (…)")`.
- the ORDER clause can optionally be specified in the property orderClauseProcess (without the word "ORDER" itself). As usual for processes, the SQL must be specified as argument of method `result.string()`, e.g., `result.string("MYTABLE.MYCOLUMN1, MYTABLE.MYCOLUMN2")`.

In order to access a database, you should use **prepared statements** instead of plain SQL code - at least in all cases, where an external input is processed (i.e., text input by the user or a variable filled by an import process, see below). Among other advantages, this increases the data security of ADITO, as attacks by the "SQL injection" technique are avoided. In the library `SqlBuilder_lib` you can find several classes providing SQL helper functions, including prepared statements, in particular, the class `SqlBuilder` and `SqlUtils`. Examples can be found in chapter SQL Helper Functions below.

B.2. Commit after database changes

All ADITO methods for inserting or changing structure or content of the database are always committed automatically. A separate "commit" statement is never required.

B.3. SQL Helper Functions

This chapter includes several examples showing

- how SQL code is applied in ADITO, using prepared statements;

- what SQL code is actually generated by various SQL helper functions.

Generally, for reasons of data security, you should **always use prepared statements** in ADITO, especially when you are processing external data, e.g.

- data input by the user (e.g., via a text field)
- data imported from an external source (e.g., the customer’s legacy system)

This prevents external attacks via "SQL injection" and, in some cases, improves the performance of the system.

The central class for building prepared statements in ADITO is `SqlBuilder`. You can find a detailed description of its usage in the property "documentation" of library `SqlBuilder_lib` (in folder process > libraries). Prerequisite for viewing this documentation in the designer is that you have the installed the plugin „AsciidoctorJ4NB“.

B.3.1. Example: contentTitleProcess of CarDriver_entity

CarDriver_entity.contentTitleProcess

```
var carDriverId = vars.get("$field.CARDRIVERID");

if (carDriverId) {

    var displayData = newSelect("SALUTATION, FIRSTNAME, LASTNAME")
    .from("PERSON")
    .join("CONTACT", "CONTACT.PERSON_ID = PERSON.PERSONID")
    .join("CARDRIVER", "CARDRIVER.CONTACT_ID = CONTACT.CONTACTID")
    .where("CARDRIVER.CARDRIVERID", carDriverId)
    .arrayRow();

    var salutation = displayData[0];
    var firstname = displayData[1];
    var lastname = displayData[2];

    result.string(salutation + " " + firstname + " " + lastname);
}
```

An instance of class `SqlBuilder` enables you to build an SQL statement using several methods, whose names are identical to the corresponding SQL clauses (select, from, join, etc.).

Method `arrayRow()` returns the SQL result as array of values: The contents of the first database result row is returned, with the column values as a one-dimensional array; possible further rows are ignored in this case.

B.3.2. Example: valueProcess of EntityField availability

Car_entity.availability.valueProcess

```
var carId = vars.get("$field.CARID");

if (carId) {

    var currentReservationId = newSelect("CARRESERVATIONID")
    .from("CARRESERVATION")
    .where("CARRESERVATION.CAR_ID", carId)
    .and("STARTDATE < CURRENT_TIMESTAMP")
    .and("ENDDATE > CURRENT_TIMESTAMP")
    .cell();

    var availability = "NO";
    if (currentReservationId == "") {
        availability = "YES";
    }
    result.string(availability);
}
```

Method `cell()` returns the SQL result as one single value: the first column value of the first row; possible further rows or columns are ignored in this case.

B.3.3. Example: conditionProcess of CarReservation_entity's RecordContainer

CarReservation_entity.RecordContainers.db.conditionProcess

```
var cond = newWhereIfSet("CARRESERVATION.CARDRIVER_ID", "$param.CarDriverId_param")
.andIfSet("CARRESERVATION.CAR_ID", "$param.CarId_param");

result.string(cond);
```

The SQL code built by the helper method `newWhereIfSet` is (if called via the MainView of Context CarDriver), e.g.:

```
CARRESERVATION.CARDRIVER_ID = '594811af-9947-4cf3-9c4c-d719cb88384a'
```

If the Parameters are not set (e.g., if you open the FilterView of Context CarReservation), the condition is an empty String (= no condition at all), so all CarReservation datasets are shown.

B.3.4. Example: Driver’s name

CarDriver_entity.db.CONTACT_ID.displayValue.expression

```
result.string(PersUtils.getResolvingDisplaySubSql("CONTACT_ID"));
```

The helper function `PersUtils.getResolvingDisplaySubSql("CONTACT_ID")` generates and returns the following SQL code, which is a subselect for displaying a person’s complete name instead of its related CONTACTID:

```
SELECT CASE
        WHEN trim(PERSON.SALUTATION) != ''
            AND PERSON.SALUTATION IS NOT NULL
            THEN CASE
                    WHEN trim(PERSON.TITLE) != ''
                        AND PERSON.TITLE IS NOT NULL
                        THEN trim(PERSON.SALUTATION) || ' '
                    ELSE trim(PERSON.SALUTATION)
                    END
        ELSE ' '
        END || CASE
        WHEN trim(PERSON.TITLE) != ''
            AND PERSON.TITLE IS NOT NULL
            THEN CASE
                    WHEN trim(PERSON.FIRSTNAME) != ''
                        AND PERSON.FIRSTNAME IS NOT NULL
                        THEN trim(PERSON.TITLE) || ' '
                    ELSE trim(PERSON.TITLE)
                    END
        ELSE ' '
        END || CASE
        WHEN trim(PERSON.FIRSTNAME) != ''
            AND PERSON.FIRSTNAME IS NOT NULL
            THEN CASE
                    WHEN trim(PERSON.MIDDLENAME) != ''
                        AND PERSON.MIDDLENAME IS NOT NULL
                        THEN trim(PERSON.FIRSTNAME) || ' '
                    ELSE trim(PERSON.FIRSTNAME)
                    END
        ELSE ' '
        END || CASE
        WHEN trim(PERSON.MIDDLENAME) != ''
            AND PERSON.MIDDLENAME IS NOT NULL
            THEN CASE
                    WHEN trim(PERSON.LASTNAME) != ''
                        AND PERSON.LASTNAME IS NOT NULL
                        THEN trim(PERSON.MIDDLENAME) || ' '
                    ELSE trim(PERSON.MIDDLENAME)
                    END
        ELSE ' '
        END || CASE
        WHEN trim(PERSON.LASTNAME) != ''
            AND PERSON.LASTNAME IS NOT NULL
            THEN trim(PERSON.LASTNAME)
        ELSE ' '
        END
FROM PERSON
JOIN CONTACT ON (PERSON.PERSONID = CONTACT.PERSON_ID)
WHERE CONTACT.CONTACTID = CONTACT_ID
```

B.3.5. Example: Manufacturer

Car_entity.db.MANUFACTURER.displayValue.expression

```
var sql = KeywordUtils.getResolvedTitleSqlPart($KeywordRegistry.carManufacturer(), "CAR.MANUFACTURER");
result.string(sql);
```

The helper function `KeywordUtils.getResolvedTitleSqlPart($KeywordRegistry.carManufacturer(), "CAR.MANUFACTURER")` returns the following SQL code:

```
CASE
        WHEN CAR.MANUFACTURER = '11a849d8-67ed-448e-86aa-6f4ab54d22ee'
            THEN 'Mercedes'
        WHEN CAR.MANUFACTURER = '107a2bf2-803a-4cf0-bf69-ff649acc113b'
            THEN 'BMW'
        WHEN CAR.MANUFACTURER = '15a66e3a-9e3e-4051-be6f-eb5b425e0fde'
            THEN 'VW'
        ELSE ''
        END
```

Appendix C: Order of execution of Entity processes

C.1. Load

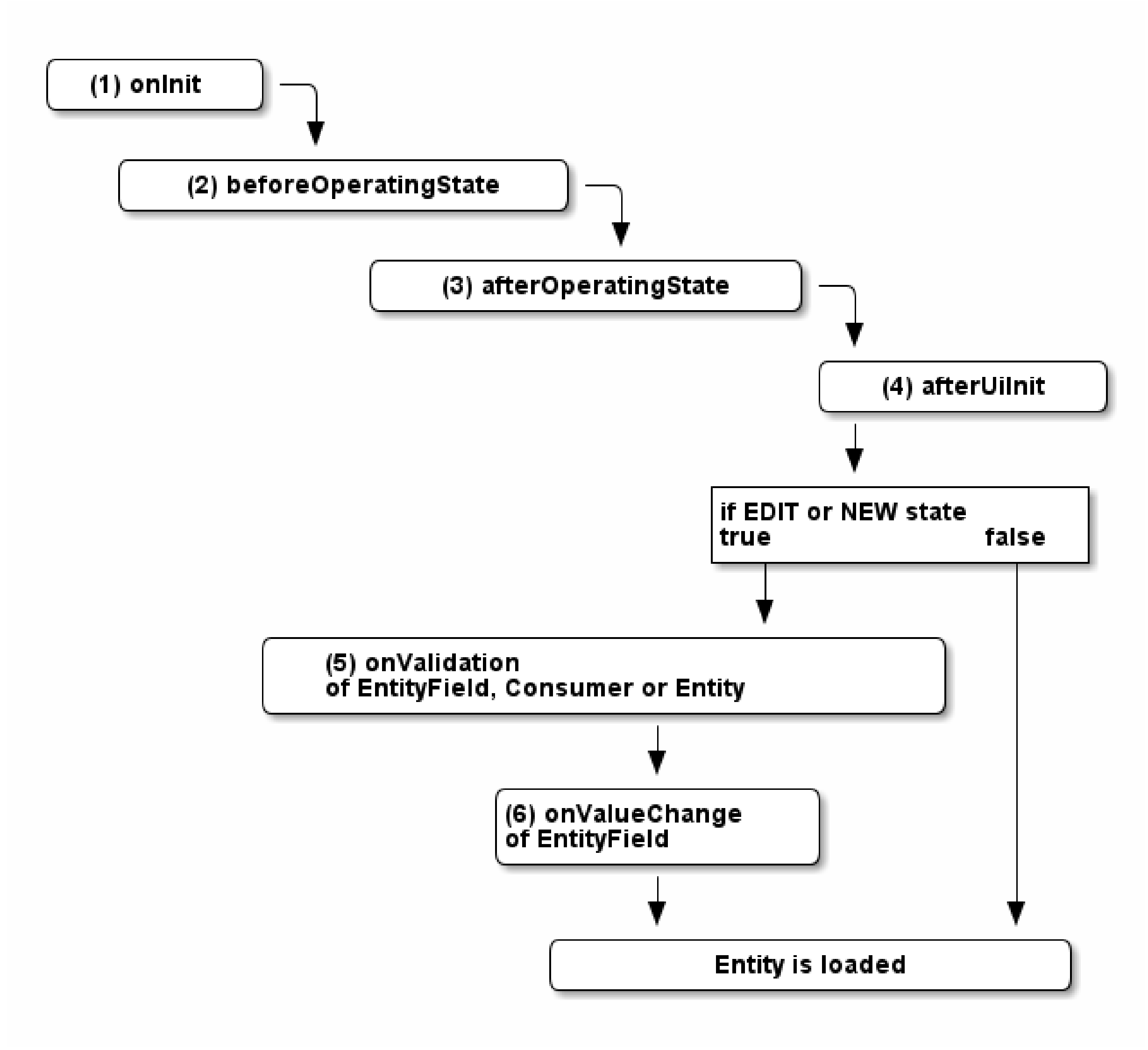


Figure 39. Order of execution of processes when loading an Entity

1. **onInit**

   This process runs before the RecordContainer is loaded. Here you can initialize variables like `$global.variablename` or `$context.variablename`.

2. **beforeOperatingState**

   When this process runs, the Entity is already loaded, but is not in one of the operating states yet. (see appendix "Operating state vs. record state")

3. **afterOperatingState**

   Here it is determined, which operating state is used, but no components are loaded yet. This is the place to react to changes in operating state.

4. **afterUiInit**
   This process runs after the UI is loaded. Here you can use methods to influence the UI, i.e. `neon.addRecord` to add a new record to an Entity.

5. **onValidation**
   In this process you can validate the data in the fields, before saving. **This process should only be used to validate data. Don't react to changes here!** If any text is given as result of the onValidation process, an automatism makes sure that

   - the validation is considered as "false";
   - the given text is displayed to the right of the "Save" button;
   - the "Save" button is disabled until the next call of the onValidation process.
   - Example: `result.string("Your input must not include special characters like '&'.");`

6. **onValueChange**
   In this EntityField process you can react to changes in an EntityField and, e.g., a computed value or set other components to inactive depending on the content.

   - With property **onValueChangeTypes** (in the EntityField's property sheet directly under property onValueChange) you can define the types of sources (modifiers) that will trigger the onValueChange process, e.g. "MASK" or "RECORD". Find more information about the selectable modifier types and their meanings in the property description of onValueChangeTypes.
   - Variable `$local.modifiertype` holds the type of the modifier (source type) that has triggered the onValueChange process - see property description of onValueChangeTypes. This variable can be helpful, if multiple modifier types had been selected in property onValueChangeTypes, and a distinctive reaction is required, depending on the modifier type.

C.2. Save

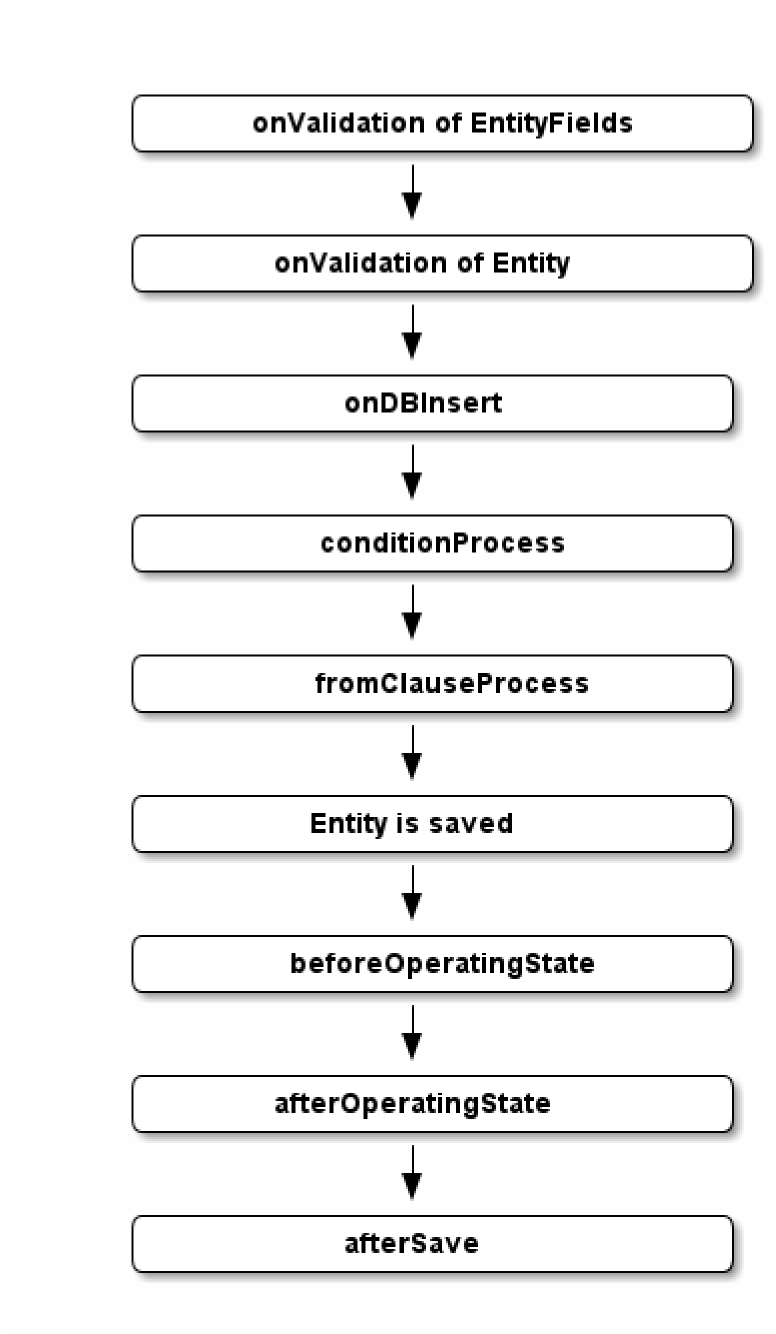


Figure 40. Order of execution of processes when saving an Entity with a dbRecordContainer

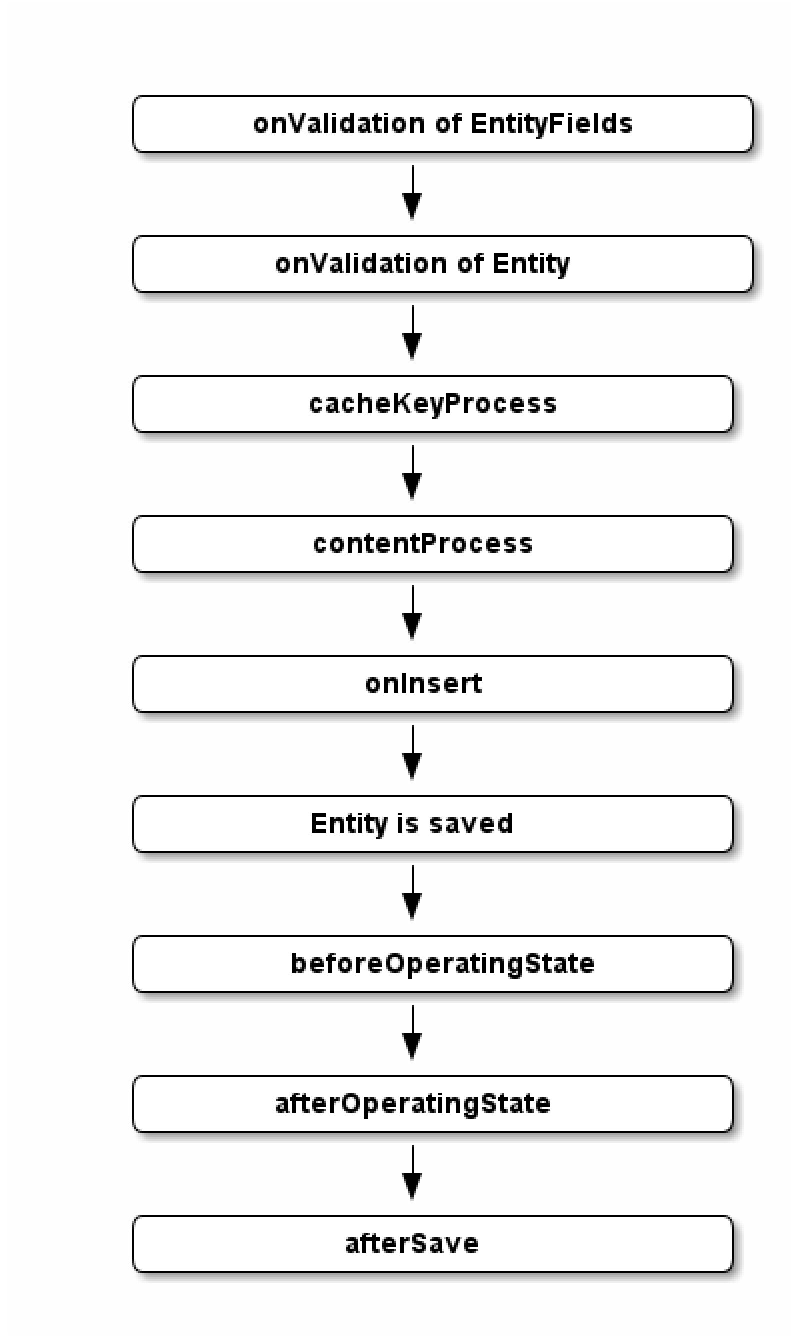


Figure 41. Order of execution of processes when saving an Entity with a jDitoRecordContainer

You can find a description of these processes in their JSDoc and partly in the previous chapter about load processes. Here is some additional information:

- **afterSave**
  The afterSave process was implemented to execute *client* commands after saving a record of an Entity. In some cases, e.g., after saving, a popup message should be shown to the user, or another Context should be opened. To try the same in the onDBInsert/onInsert process, leads to problems:

- Client commands (`neon.xxx`, `question.xxx`) are not allowed to be executed in RecordContainer processes. (Entity can also be used server-side with "Read/Write Entity" methods, then the methods lead to errors.)
- `openContext` does not work or was "overwritten".

  Therefore, the afterSave process was introduced, in order to allow the execution of these kind of actions after saving.

- Please note the following:
  - The afterSave process is exclusively executed client-side. Therfore, in this process, nothing must be changed/triggered etc. (processes, updates etc.). If this is nevertheless necessary, it must be done manually on the server side, e.g., in the on(DB)Insert process.
  - If a new Context is opened in the afterSave process, `result.string(true);` must be returned, in order to avoid the default behavior after saving.

Appendix D: Requirements for customized Theme

In the below table you can find all information required by ADITO's development department and its UX designer in order to create a customized Theme that is optimized for being compliant with the Display Screen Equipment Directive (e.g., contrast effects).

Table 5. Requirements for the creation of a customized Theme

Element	**Location of usage**	**Default**
Background image	Login-/Logout View	ADITO image
Company logo	Login-/Logout View	ADITO logo
First main color	Buttons, hyperlinks, focus color	Dark blue
Second main color	Menu header, active menu entry	Red
1 to 20 user-specific colors	Charts, Avatars, score cards	Various

Alternatively, if available, you can send us your own CI guide.

You can find extensive background information on the topic "Themes" in the ADITO Information Document AID121 "Themes".

It is strictly against the intention of ADITO that users modify the Theme by themselves. It is **exclusively** ADITO's development department that is authorized to modify a Theme or create a new Theme.

Appendix E: Checklist for new fields

Here is a checklist for how to proceed if you want to add a new EntityField. First of all,

- inform your ADITO client administrator about the new field, in order to make sure that its access rights are configured correctly.
- inform the data security official in charge with your project (e.g., in Germany, the "Datenschutzbeauftragter") about the new field, in order to make sure that possible concerns will be included in the further configuration and programming (e.g., the implementation of a dialog pointing to the "impact on the data privacy information (GDPR)" - see, e.g., method DataPrivacyUtils.notifyNeedDataPrivacyUpdate in DataPrivacy_lib).

Technically, the new field is added as follows:

- In the "Projects" window, double-click on the Entity, so it is shown in the Navigator window.
- In the Navigator window, right-click on the sub-node "Fields" and choose "New Field".
- Enter the field’s name - respecting the ADITO spelling guidelines (see ADITO Information Document AID001, chapter "Spelling & Wording" > "ADITO models") - and press OK.
- Configure the new field’s properties:
  - "title": Enter a title to be shown with the new field in the client, e.g., as label or as table column header.
  - "contentType": Change default value "TEXT", if required. For type "DATE", set also property "resolution".
  - Set further properties, if required (such as valueProcess, for calculated fields).
- (If the field’s value is to be stored in the database:) Create a new column corresponding to the field, via one of the following ways:
  - Use the ADITO database editor (system > default > Data_alias > ADITO): Right-click on the database table and choose "Add column…" from the context menu. Make sure the name of the new column is spelled exactly like the name of the EntityField. Enter all required column properties, e.g., the data type. Consequently, update the Alias Definition (double-click on alias > Data_alias, then, in the Navigator window, choose "Diff Alias <> DB Table" from the context menu of parent node "Data_alias", and perform an update from "remote" to "local").
  - Alternatively, you can do it the other way round: Add the column in the Alias Definition: Double-click on alias > Data_alias. Then, in the Navigator window, right-click on the database table and choose "New Column" from the context menu. Make sure name of the new column is spelled exactly like the name of the EntityField. Configure all required

column properties, e.g., "columnType". Consequently, update the database (in the Navigator window, choose "Diff Alias <> DB Table" from the context menu of parent node "Data_alias", and perform an update from "local" to "remote"). If you have a Liquibase create file for this table, add the column manually.

  - If you prefer to use Liquibase instead, create a separate change set for adding the column. You can use one of the "alter_xxx.xml" files in one of the folders under alias > Data_alias > basic as pattern, e.g., the file alter_SerialLetter.xml (under alias > Data_alias > basic > 2019.2.1). +

```
<?xml version="1.1" encoding="UTF-8" standalone="no"?>
<databaseChangeLog xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
xmlns:ext="http://www.liquibase.org/xml/ns/dbchangelog-ext" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog-ext
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-ext.xsd http://www.liquibase.org/xml/ns/dbchangelog
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.6.xsd">
    <changeSet author="j.smith" id="77bf2086-3eac-4a21-bb03-168140477e19">
        <addColumn tableName="MYTABLE">
            <column name="MYCOLUMN" type="NVARCHAR(50)"/>
        </addColumn>
    </changeSet>
</databaseChangeLog>
```

    Perform a Liquibase update (right-click on alias > Data_alias and choose "Liquibase > Update…" from the context menu). Remember to reference this new xml file also in the appropriate changelog.xml file, making sure that it is executed *after* the create file of the respective table. Consequently, update the Alias Definition (double-click on alias > Data_alias, then, in the Navigator window, choose "Diff Alias <> DB Table" from the context menu of parent node "Data_alias", and perform an update from "remote" to "local").

- (If the field's value is to be stored in the database:) Establish the connection between the field and the corresponding database column:
  - Set the new column in property "recordfield" of the new EntityField's corresponding ".value" RecordFieldMapping in the RecordContainer.
  - If another value than the stored value is to be displayed:
    - Enter a subselect for the display value in property "expression" of the new EntityField's corresponding ".displayValue" RecordFieldMapping in the RecordContainer, and/or
    - enter a code retrieving the display value in the displayValueProcess of the new field.
- Add the new field in the "columns" or "fields" properties of the ViewTemplates of all Views in which the new field is to be shown.
- Include all configurations and code required to ensure the correct access rights and all concerns

of your data security official (see above).

- Save and deploy.
- Clear the cache: Open the ADITO Manager (in the client’s Global Menu, choose "Server" in menu group "Manager"), mark your server and choose option "Clear cache" via three-dotted button in the PreviewView. Alternatively, of course, you can achieve the same effect by re-starting the server.
- Re-login to the ADITO client.

Appendix F: Accessing the value of an EntityField

This appendix is about the relation between the value of an EntityField and its associated variables.

> In this chapter, "value of an EntityField" is actually referring to a specific value processed in the ADITO core. For a better understanding of the following explanations, you can simply consider "value of an EntityField" to be the value that is visible in the client.

If you have created an EntityField called MYFIELD, then it automatically has a system variable with the name `$field.MYFIELD` associated. You can access the value of this variable via the method `vars.get("$field.MYFIELD")`.

The EntityField value and the "$field" variable value are linked in the ADITO core. Each is calculated at different times, and at certain points they are synchronized to each other.

F.1. Synchronization

There are two ways of synchronizing:

1. **EntityField value → "$field" variable value**

   If the value of an EntityField is set, then it triggers a new calculation of the variable.

2. **"$field" variable value → EntityField value**

   If the value of the "$field" variable changes, its value gets set to the EntityField value.

F.2. How does an EntityField value get set?

The value of an EntityField is set, when a user enters a value in a View. This is when the new calculation of the variable is triggered according to the order of processes detailed in a previous appendix.

F.3. How does a "$field" variable get its value?

There are three variants of how the value of a "$field" variable can be determined:

1. **Record**
   If you have linked your EntityField with your RecordContainer and have no valueProcess specified, then the system takes the value from the record.
2. **Process**
   If your EntityField is not linked to the RecordContainer, then the valueProcess is used to determine the value of the variable.

3. **Record process**
   This means, you have a mixture of the previous variants: You have linked your EntityField within the RecordContainer **and** you have a valueProcess specified.
   If the record returns a value, then it is used, while the valueProcess is being ignored. If, however, the record does not return a value, then the valueProcess is executed to determine a value.

Please mind the following logic:

- If your system changes into state "NEW" or "EDIT" (e.g., if you open the EditView of a Context), **all** fields of the Context's Entity are being retrieved (= loaded or calculated) - no matter if they are displayed or only used in other processes. This makes sure that all data are up-to-date when the user edits or enters a dataset.
- Generally: If you open a View (regardless of the system's state), ADITO automatically determines, what fields **might** be required - meaning not only the fields referenced in the View configuration, but also fields that are used in one of the Entity's processes, like, e.g., the titleProcess or the onActionProcess. These fields are then always loaded (= the column or the "expression" of the RecordFieldMapping will be included in the SQL's SELECT clause) - even if it later turns out that one or several fields' values are actually not used.
  This will ensure that all (possibly) required values are immediately present when the client user works with the View - without the system having to load/calculate them separately. On the other hand, this can lead to performance issues, if there are a lot of fields calculated via property "expression" - especially when these fields' calculation is complex or suboptimally realized.
  On the contrary, if the field is only calculated via a valueProcess or displayValueProcess (and not via the "expression"), the calculation is only done on demand.
  If both types of calculation have been configured, the "expression" is automatically preferred, if it actually returns a value.

Furthermore, the determination of the "$field" variable's value depends on the state of the record:

1. **VIEW mode**

In VIEW mode, the user is only Viewing the data and cannot change it. In this mode, the value is determined by the previously stated variants.

1. **EDIT mode**
   To to determine a value in EDIT mode, the system proceeds as follows:
   - First, it checks if a valueProcess is specified. If so, then it is executed. Thus, the valueProcess can be used to preset a value. (This is where the `$this.value` variable has to be used (see next chapter).
   - If the valueProcess does not return a value, then the user's input is used.

F.4. $this.value

`$this.value` is a special system variable that is accessible in the valueProcess of an EntityField. It is accessed via `vars.get("$this.value")` and contains the current value of the EntityField (being input or preset). It can therefore, e.g., be used to determine if something was entered or if the field was completely empty.

> If the field has no current value, `$this.value` contains `null`, otherwise it contains the value entered by the user.
> If the user has deleted the value, `$this.value` will return an empty string (no longer `null`!).

Therefore, if you want to preset a value, you have to check for `if( vars.get("$this.value") == null )` to only set your preset value if no value was present before. In the onValidation and onValueChange processes, you can use this to get and process the users input.

Example:
Presetting currency to "EUR":

Productprice_entity.CURRENCY.valueProcess.js

```
if (vars.get("$sys.recordstate") == neon.OPERATINGSTATE_NEW && vars.get("$this.value") == null)
{
    result.string($KeywordRegistry.currency$eur());
}
```

F.5. $this.value vs. $field.MYFIELD

While variable `$this.value` contains the current value of a field, variable `$field.<FIELDNAME>` contains the last calculated value that was synchronized to the field. In most cases, those two variables have the same value, due to a close synchronization. There are rare exceptions when the values can differ, e.g.

- while the initial value of a field is determined;
- when a record gets reloaded.

F.6. $this.value and $field.MYFIELD in valueProcess

Generally, in a valueProcess you have to distinguish between the stored field value ($field.MYFIELD) and the new value to set ($this.value).

Here is an overview about when and how a valueProcess is executed and what reactions are possible:

1. Initial loading of the field values
   If the value of the field is initially loaded from the Entity, then $this.value is null. This case is mostly used for presets when entering new data oder editing it, because this case occurs only once. If you miss to check for `$this.value == null` then the value of the field will be overwritten with every refresh.

2. Field is explicitely set empty
   In case the field is explicitely set empty (e.g., by the user), a check for `!this.value` would fail, because in this case `$this.value == ""`

3. Changes of the field itself
   If the value of the field itself changes directly (e.g., by user input, WriteEntity, `neon.setFieldValue`, etc.) then `$this.value` is filled with a value (or an empty string in case 2, see above) and the field itself (`$field.MYFIELD`) ist empty (`""`). Knowing this, you can, e.g., prevent that the field is automatically filled by dependencies from other fields, because the given value was entered explicitely.

4. Trigger of the valueProcess by other fields
   This case only happens when entering new data, not when editing it. It is the constellation that the value of the field itself has not changed, but it has been updated because of triggers/calls etc. and thus the valueProcess is executed. In this case `$this.value` and `$field.MYFIELD` have the same value. This constellation can be used, e.g., to set the field depending on other fields. This is the recommended approach, other than to use `neon.setFieldValue` in the onValueChange process of another field.

> `neon.setFieldValue` should only be used in onValueChange, if there is absolutely no other possibility to realize the task. The performance of `neon.setFieldValue` is very low, because many dependencies need to be updated and it can happen that the same code needs to be written in multiple fields.

If the value of a field is set and you do not return anything in the valueProcess, then the set value will be used. (This means, in case 3 only the return must be prevented.)

Example of the implementation of the above cases 1, 3, and 4:
The example task is that a field A should initially be filled with value 1, when entering a new dataset. It should be possible to overwrite the field’s value when entering a new dataset or when editing it. If field B is set to a specific value, then field A should automatically filled with the value 2 (works for "new", not for "edit").

Example valueProcess for cases 1, 2, and 4

```
var fieldA = vars.get("$field.A");
var thisValue = vars.get("$this.value");

// The value of the field is not required in this case,
// but it can work as trigger, if B changes.
var fieldB = vars.get("$field.B");

var recordState = vars.get("$sys.recordstate");

if([neon.OPERATINGSTATE_NEW, neon.OPERATINGSTATE_EDIT].includes(recordState))
{
    if(recordState == neon.OPERATINGSTATE_NEW && thisValue == null)
    //case 1: initial presetting the field with value 1
    {
        result.string(1);
    }
    else if(fieldA == "" && thisValue)
    //case 3: value was changed -> should be set now
    {
        result.string(thisValue);
        // In this case, you can alternatively simply return nothing.
        // Then $this.value will be set as field value.
    }
    else if(fieldA == thisValue)
    // case 4: thisValue and field have the same value
    // -> The field was not changed, but was triggered somewhere,
    // e.g., because a change of field B -> fieldA should be set to 2
    {
        result.string(2);
    }
}
```

F.7. $local.value

This is a local system variable that is accessible in the onValidation and the onValueChange processes and contains the entered value **before** it is written to the variable value, so you can validate it before the data enters the system.

Example:
Checking if the user has input a wrong entry date:

Activity_entity.ENTRYDATE.onValidation.js

```
var entryDate = vars.get("$local.value");
if (!DateUtils.validateNotInFuture(entryDate)) {
    result.string(translate.text("Entrydate must not be in the future"));
}
```

F.8. $local.rowdata and $local.initialRowdata

If you want to access the values of EntityFields in specific processes of a RecordContainer, you must exclusively use `$local.rowdata` or `$local.initialRowdata`, because `$field` variables might contain outdated values at that time. In particular, these are the following processes:

- dbRecordContainer: onDBInsert, onDBUpdate, and onDBDelete
- jDitoRecordContainer: onInsert, onUpdate, and onDelete (see also sub-chapter "Advanced explanations" of chapter JDitoRecordContainer)

Find more information on `$local.rowdata` and `$local.initialRowdata` in appendix $local variables.

Appendix G: Operating state vs. record state

In ADITO, there are 2 types of system states, which can be retrieved via system variables:

- "operating state" → vars.get("$sys.operatingstate")
- "record state" → vars.get("$sys.recordstate")

The values of these state variables are Strings, namely (both for operating state and record state)

- "VIEW"
- "NEW"
- "EDIT"
- "SEARCH"

When coding and checking a state for a specific value, you should not use the above Strings itself, but the corresponding constants

- `neon.OPERATINGSTATE_VIEW`
- `neon.OPERATINGSTATE_NEW`
- `neon.OPERATINGSTATE_EDIT`
- `neon.OPERATINGSTATE_SEARCH`

These constants are used both for operating state and for record state. There are no specific constants for record state.

Operating state and record state are similar, but apply in different environments:

- Operating state refers to the state a Context is currently in, while
- record state refers to the state of a record or even only a part of a record.

Example:
If you are in a MainView and use the pencil button of a component, the Context is still in the operating state "VIEW", while the part of the record, which is covered by the component, is in record state "EDIT".

Another example:
If you are in an EditView, operating state is "EDIT", because the whole Context is now in the "EDIT" mode. The record state is now also "EDIT", because the record is edited with the help of the EditView.

In most cases, you should use `$sys.recordstate` to determine if data is being edited, as the operating state still can be the VIEW mode, while the record state is EDIT.
Use `$sys.operatingstate` to determine if you are in an EditView.

Code example:

```
//Presetting an ID, if a record is newly created.
if( vars.get("$sys.recordstate") == neon.OPERATINGSTATE_NEW && vars.get("$this.value") == null )
{
    result.string( util.getNewUUID() );
}
```

Appendix H: LoadEntity and WriteEntity

LoadEntity and WriteEntity are essential functionality of the ADITO platform, used to manage datasets/records. The naming of these terms is not related to method names, but they summarize functionality represented by several methods used for loading and writing data from/into an Entity.

This means, the loading/writing does not, in the first place, target the database, but the Entity and its Fields - respecting customized logic for values, displayValues, validations, etc. Of course, at last, the database will be accessed via the Entity, but also calculated EntityFields, without relation to the database, can be accessed.

You primarily use LoadEntity and WriteEntity if you want to

- load and write datasets strictly according to the permissions (access rights) configured by the client administrator - which, e.g., is not the case when loading data via SqlBuilder or the db.xxx methods (!).
- load or write data which is related to an Entity that is based on more than one database table (no need to care for SQL joins etc.).
- load a calculated EntityField
- load the displayValue of an EntityField
- have an EntityField’s presets, validations, and dependencies to be respected in the loading/writing logic
- utilize data caching via a RecordContainerCache.

LoadEntity and WriteEntity should be your preferred way to load and write data in ADITO, when permissions (access rights) must be respected. Whenever you use other loading/writing options (e.g., SqlBuilder, db.xxx methods, etc.), the permissions configured by the client administrator will be ignored, which may cause severe data security leaks, as critical data may be disclosed to unauthorised persons.

However, performance issues can be involved, as LoadEntity and WriteEntity include more overhead and calculations than an SQL select or insert. Therefore, it is important to be careful when utilizing LoadEntity and WriteEntity: If the code is frequently executed within loops or recurring processes and requires optimal speed, LoadEntity and WriteEntity may not be the most suitable options. Particularly in Entity processes, LoadEntity and WriteEntity calls should be handled carefully, e.g., in onValidation, stateProcess, valueProcess, or displayValueProcess. Problems can occur, because

- processes of the Entity are executed when loading
- a count is executed when loading
- When larger data amounts are involved, LoadEntity consumes extensively RAM.

Example: If you use LoadEntity for validation of input of a Consumer, then the already saved connected data will be loaded via LoadEntity in the onValidation process of the Consumer. This process will be executed very often during editing, and one Entity loading can take from 0,5 to up to multiple seconds. In this case, the required data is very small, and in most cases there is no need for respecting permissions. Thus, an SQL select would do the same job in a fraction of the time LoadEntity requires.

In order to use any method of LoadEntity or WriteEntity, we need the following import:

```
import { entities } from "@aditosoftware/jdito-types";
```

The documentation (JSDoc) of each method (available by using CRTL+SPACE) is not finished yet. It will be available in a future ADITO version.

H.1. LoadEntity

The term LoadEntity summarizes the methods to get datasets related to an Entity.

The first step is to create a configuration object. There are 2 types of configuration objects available, specific for different purposes:

```
// configuration for loading datasets
// (return value: Object of type "LoadRowsConfig")
var myConfig1 = entities.createConfigForLoadingRows()

// configuration for loading datasets from an Entity via a Consumer
// (return value: Object of type "LoadConsumerRowsConfig")
var myConfig2 = entities.createConfigForLoadingConsumerRows()
```

These configuration objects provide setter methods (parameters) that can be chained in order to define the datasets that should be loaded (similar to the chaining approach of, e.g., SqlBuilder). The order of the method calls does not matter. To see how to add these parameters, please refer to the example below. By adding a "." to the end of the configuration you can use the code completion to see all available functions by using CTRL+SPACE.

Table 6. All setter methods (parameters) available for LoadEntity

Setter Method	Description
`.entity` (String)	the name of the Entity whose datasets are to be loaded
`.fields` (Array)	list of EntityFields of the Entity. If you specify here only "#UID", then the ADITO system tries to optimize the loading process (e.g., by skipping unnecessary processes)
`.filter` (String)	filter to be applied when loading the datasets
`.count` (Number)	maximum number of datasets
`.provider` (String)	name of the Provider that is to be used to retrieve the data
`.addParameter` (String, String)	specifies a Parameter, to be used to supply a Provider. The first argument of this method is the name of the Parameter (e.g., "ContactId_param"), the second argument is the value to be assigned to this Parameter, e.g., a UID).
`.startRow` (Number)	number of the row of the datasets to start loading
`.uid` (String)	UID of the dataset to be loaded
`.uids` (Array)	UIDs of the datasets to be loaded. **NOTE:** If the argument of this setter method is • an empty Array, then *nothing* is loaded (subsequent method `getRows` returns an empty Array, and method `getRowCount` returns 0) • null, then there there is *not any UID-related restriction at all* (= same as if this setter method was not executed at all)
`.user` (String)	title of the user (e.g. "Harold Smith"), in whose "user context" (permissions, calendar or mail settings, etc.) the loading logic will be executed. For reasons of data security, this works only in server processes. (In client-related processes, it will cause an error.)

`.ignorePermissions()`	Load without respecting the permissions of the respective user (but other user-specific functionality, e.g., calendar or mail settings, do still apply).

The methods `.entity` and `.fields` are mandatory. If these are not used, the configuration is invalid. You can see the documentation (JSDoc) of each method by using CRTL+SPACE.

The configuration can be executed via 3 different methods, which have different purposes.

Table 7. The executing methods of LoadEntity

Method	Description
`entities.getRow` (LoadRowsConfig)	returns a single row (datasets)
`entities.getRows` (LoadRowsConfig)	returns all rows (datasets)
`entities.getRowCount` (LoadRowsConfig)	returns the number of rows (datasets)

(The methods for LoadConsumerRowsConfig are the same.)

You can see the documentation (JSDoc) of each method by using CRTL+SPACE.

The methods `entities.getRow` and `entities.getRows` differ in their behavior, which has an effect especially on the processing of the results and the filling of variables like `sys.uid` - see chapter getRow vs. getRows.

H.1.1. Benefits

Using LoadEntity shows the following advantages compared to loading data directly via SqlBuilder or the db.xxx methods:

1. LoadEntity respects the permissions (access rights) configured by the client administrator. Nevertheless, if required, you can skip the permissions, by adding `.ignorePermissions()`.
2. Complex SQL queries (with JOINs, subselects, etc.) can be avoided - e.g., in cases when an Entity is related to more than one single database table.
3. An EntityField's presets and dependencies are respected in the loading logic.
4. The data is loaded via the RecordContainer of the Entity; thus, all data can be cached - which results in a better user experience and faster response times when using programs like, e.g., Apache Ignite.

5. Every EntityField of the Entity can be loaded, even if this EntityField is not directly related to one specific database field (e.g., a calculated EntityField).

H.1.2. Example

Below you find an example code of a test Action that loads all datasets of the xRM project’s Entity Activity_entity and logs the result in detail. The loading is restricted to the values of the fields SUBJECT, INFO, ENTRYDATE, and the display values of the fields DIRECTION and RESPONSIBLE. You can always reduce the size of the result set by filtering according to values or entering UIDs.

MyTest_entity.testAction1a.onActionProcess

```
// creating the configuration object
var config = entities.createConfigForLoadingRows();
// setting the Entity's name
config.entity("Activity_entity");
// defining the required EntityFields
config.fields([
    "SUBJECT",
    "INFO",
    "DIRECTION.displayValue",
    "ENTRYDATE",
    "RESPONSIBLE.displayValue"
    ]);

// optional restriction to 1 UID
// config.uid("0cf02b72-a46a-4cd2-975f-15556618ea90");

// optional restriction to multiple UIDs
// config.uids(["0cf02b72-a46a-4cd2-975f-15556618ea90",
// "21852330-9c66-42a3-9d25-d053833f146d"]);

var myResult = entities.getRows(config);

// Retrieving a summary of each dataset
for (let i in myResult)        {
            logging.log("-----> Dataset number " + i + ":")
            logging.log(myResult[i]);
        }

// Retrieving the single values of specific EntityFields
for (let i = 0; i < myResult.length; i++) {
    logging.log("-----> Dataset number " + i);
    // each part of the result is an associative array
    logging.log("SUBJECT = " + myResult[i]["SUBJECT"]);
    logging.log("INFO  = " + myResult[i]["INFO"]);
    logging.log("DIRECTION = " + myResult[i]["DIRECTION.displayValue"]);
    logging.log("ENTRYDATE  = " + myResult[i]["ENTRYDATE"]);
    logging.log("RESPONSIBLE = " + myResult[i]["RESPONSIBLE.displayValue"]);
}
```

Variation: Example of loading only 1 specific dataset via `entities.getRow(config)`. .MyTest_entity.testAction1b.onActionProcess

```
var config = entities.createConfigForLoadingRows();
config.entity("Activity_entity");

config.fields([
    "SUBJECT",
    "INFO",
    "DIRECTION.displayValue",
    "ENTRYDATE",
    "RESPONSIBLE.displayValue"
    ]);

// Restriction to 1 UID
config.uid("0cf02b72-a46a-4cd2-975f-15556618ea90");

var myResult = entities.getRow(config);
// Retrieving each field of the dataset
for (let i in myResult) {
    // e.g., myResult["SUBJECT"]
    logging.log("-----> Dataset index = " + i + ": " + myResult[i]);
}
```

Please note that, in this case, the result is 1 single object, which you can directly access as associative array, e.g., like this: `myResult["SUBJECT"]`. Furthermore, note that `entities.getRow(config)` requires a configuration that restricts the result to one single dataset. Otherwise, you will get an exception.

H.1.3. getRow vs. getRows

The methods `entities.getRow` and `entities.getRows` differ in their behavior, which has an effect especially on the processing of the results and the filling of variables like `sys.uid`.

`entities.getRow`:

- This method loads a specific dataset.
- Variables like `sys.uid` are automatically filled with the values of the loaded dataset.
- If the requested dataset cannot be found, an exception is thrown, which must explicitly be caught by an individual error handling.
- The behavior is similar to opening a PreviewView or MainView.

`entities.getRows`:

- Here, multiple datasets are returned, based on a filter.
- Variables refering to single datasets, like `sys.uid`, are not filled.
- If no datasets are found, no exception will be thrown - even not in case only one single dataset was expected.
- The behavior is similar to the loading of a FilterView.

Consequences in practice:

- **Specific data handling:** If it is required that variables like `sys.uid` are filled, then `entities.getRow` should be used. In this case, you need to make sure that possible exceptions (caused, e.g., by not findable datasets) are handled appropriately.
- **Exception-tolerant queries:** `entities.getRows` should be used for queries with no exact number of hits to be guaranteed or expected, in order to avoid exceptions and to keep results flexible.

Example:

Here is an example code to be used with `entities.getRow`, covering every error case:

```
var conf = entities.createConfigForLoadingRows()
    .entity("Person_entity")
    .uid("38cb4fab-64f9-4d8e-aa6f-a158d13fc933")
    .fields(["#CONTENTTITLE"]);

try {
    var myRow = entities.getRow(conf);
    log.info("Dataset successfully loaded: " + myRow["#CONTENTTITLE"]);

    // process dataset
    (...)
    }
} catch (exception) {
    // Exception handling (can be adapted to requirements individually)
    log.error("Error when loading dataset: " + exception.message);

    // Specific action in case of an exception
    // e.g., setting standard values, informing the user,
    // or cancelling the operation
    (...)
}
```

H.2. WriteEntity

The term WriteEntity summarizes the methods to write datasets "into" an Entity, and thus, into the database table(s) related to it (create, update, or delete records).

The first step is to create a configuration. There are 3 types configurations available, specific for different purposes:

```
// configuration for creating new datasets
var myConfig = entities.createConfigForAddingRows()

// configuration for updating new datasets
var myConfig = entities.createConfigForUpdatingRows()

// configuration for delecting new datasets
var myConfig = entities.createConfigForDeletingRows()
```

You can see the documentation (JSDoc) of each method by using CRTL+SPACE.

These configuration objects provide setter methods (parameters) that can be chained in order to define the datasets that should be loaded (similar to the chaining approach of, e.g., SqlBuilder). The order of the method calls does not matter. To see how to add these parameters, please refer to the example below. By adding a "." to the end of the configuration you can use the code completion to see all available functions by using CTRL+SPACE.

Depending on the configuration type, there are different parameters available:

Table 8. All setter methods (parameters) available for **create** *configuration (createConfigFor* **Adding***Rows())*

Setter Method	Description of arguments
`.entity` (String)	the name of the Entity whose datasets are to be written
`.fieldValues` (Array)	an Array of the EntityFields or Consumers, along with their values (Important: Mind the order! See information box further below.) If the values are restricted by a value list (via dropDownProcess or a Consumer) there is no validation, i.e., the values are written as given, even if they are not included in the value list. If you need a validation, use onValidation.
`.consumer` (String)	name of the Consumer that is to be used to write the data

`.provider` (String)	name of the Provider that is to be used to write the data
`.addParameter` (String, String)	specifies a Parameter, to be used to supply a Provider or a Consumer. The first argument of this method is the name of the Parameter (e.g., "ContactId_param"), the second argument is the value to be assigned to this Parameter, e.g., a UID).
`.user` (String)	title of the user (e.g. "Harold Smith"), in whose "user context" (permissions, calendar or mail settings, etc.) the create logic will be executed. For reasons of data security, this works only in server processes. (In client-related processes, it will cause an error.)
`.ignorePermissions()`	Write without respecting the permissions of the respective user (but other user-specific functionality, e.g., calendar or mail settings, do still apply).

For the create configuration, the methods `.entity` and `.fieldValues` are mandatory. If these are not used, the configuration is invalid.

When writing the Array-typed argument of method `.fieldValues`, please urgently consider the correct **order** of the EntityFields, as the ADITO platform will process the EntityFields exactly in the given order. This is crucial, if one EntityField is logically dependend on another EntityField - e.g., if the valueProcess of MYENTITYFIELD2 contains the code `vars.get("$field.MYENTITYFIELD1")`, then, in the Array, MYENTITYFIELD1 must necessarily be specified *before* MYENTITYFIELD2. Otherwise, the required value of MYENTITYFIELD1 will not yet be set when `vars.get` is called. This behavior no bug, but intended, because WriteEntity should work like a user works in the client: If, e.g., users call an Action without filling in the value of a dependent EntityField before, they will also not get the intended result.

You can see the documentation (JSDoc) of each method by using CRTL+SPACE.

Table 9. All setter methods (parameters) available for **update** *configuration (createConfigFor* **Updating***Rows())*

Setter Method	Description
`.entity` (String)	the name of the Entity whose datasets are to be written

`.uid` (String)	UID of the dataset to be updated
`.fieldValues` (Array)	an Array of EntityFields or Consumers, along with their values (Important: Mind the order! See information box above.) If the values are restricted by a value list (via dropDownProcess or a Consumer) there is no validation, i.e., the values are written as given, even if they are not included in the value list. If you need a validation, use onValidation.
`.consumer` (String)	name of the Consumer that is to be used to update the data
`.provider` (String)	name of the Provider that is to be used to update the data
`.addParameter` (String, String)	specifies a Parameter, to be used to supply a Provider or a Consumer. The first argument of this method is the name of the Parameter (e.g., "ContactId_param"), the second argument is the value to be assigned to this Parameter, e.g., a UID).
`.user` (String)	title of the user (e.g. "Harold Smith"), in whose "user context" (permissions, calendar or mail settings, etc.) the update logic will be executed. For reasons of data security, this works only in server processes. (In client-related processes, it will cause an error.)
`.ignorePermissions()`	Write without respecting the permissions of the respective user (but other user-specific functionality, e.g., calendar or mail settings, do still apply).

For the update configuration, the methods `.entity`, `.fieldValues`, and `.uid` are mandatory. If these are not used, the configuration is invalid.

You can see the documentation (JSDoc) of each method by using CRTL+SPACE.

Table 10. All setter methods (parameters) available for **delete** *configuration (createConfigFor* **Deleting***Rows())*

Setter Method	Description
`.entity` (String)	the name of the Entity whose datasets are to be written

`.uid` (String)	UID of the dataset to be updated
`.provider` (String)	name of the Provider that is to be used to delete the data
`.addParameter` (String, String)	specifies a Parameter, to be used to supply a Provider. The first argument of this method is the name of the Parameter (e.g., "ContactId_param"), the second argument is the value to be assigned to this Parameter, e.g., a UID).
`.user` (String)	title of the user (e.g. "Harold Smith"), in whose "user context" (permissions, calendar or mail settings, etc.) the delete logic will be executed. For reasons of data security, this works only in server processes. (In client-related processes, it will cause an error.)
`.ignorePermissions()`	Load without respecting the permissions of the respective user (but other user-specific functionality, e.g., calendar or mail settings, do still apply).

For the delete configuration, the methods `.entity` and `.uid` are mandatory. If these are not used, the configuration is invalid.

You can see the documentation (JSDoc) of each method by using CRTL+SPACE.

Depending on the purpose (and thus, on the configuration), there are the following execute methods:

Table 11. The execute methods of WriteEntity

Function	Description
`entities.createRow` (CreateRowConfig)	creates a new dataset and returns its UID
`entities.updateRow` (UpdateRowConfig)	updates the dataset
`entities.deleteRow` (DeleteRowConfig)	deletes the dataset

You can see the documentation (JSDoc) of each method by using CRTL+SPACE.

H.2.1. Benefits

Using WriteEntity shows the following advantages compared to writing, updating, or deleting data directly via SqlBuilder or the db.xxx methods:

1. WriteEntity respects the permissions (access rights) configured by the client administrator. Nevertheless, if required, you can skip the permissions, by adding `.ignorePermissions()`.
2. Complex or multiple SQL queries can be avoided - e.g., in cases when an Entity is related to more than one single database table.
3. Updated or deleted records can be cached, which results in a better user experience and faster response times when using programs like, e.g., Apache Ignite.
4. An EntityField's presets, validations, and dependencies are respected in the writing logic.
5. No need for subsequently executing refresh logic (like `neon.refreshAll()`). This means, e.g., in a "table" ViewTemplate
   a. `deleteRow` only deletes the respective datasets - no refreshing/reloading of all datasets required.
   b. `updateRow` automatically reloads (only) the respective datasets.
6. Every EntityField of the Entity can be loaded, even if this EntityField is not directly related to one specific database field (e.g., a calculated EntityField).
7. Encapsulation with configurations.

H.2.2. Examples

Example 1:

Below you find an example code of a test Action that creates an Activity dataset, without ActivityLinks.

MyTest_entity.testAction2a.onActionProcess

```
// creating the configuration object
var config = entities.createConfigForAddingRows();

// name of the Entity
config.entity("Activity_entity");

// mapping of the EntityFields and their values
config.fieldValues({
    "SUBJECT": "Test Activity",
    "INFO": "This is some demo information",
    "DIRECTION": "o",
    "ENTRYDATE": datetime.date().toString(),
    "CATEGORY": "MAIL"
});

// execution method for creating a new dataset
var id = entities.createRow(config);
```

```
// loggin the automatically created UID of the new dataset,
// e.g., "38cb4fab-64f9-4d8e-aa6f-a158d13fc978"
logging.log("ACTIVITYID: " + id);
```

After executing this Action’s code, the "onDBInsert" process of the given Entity will be executed.

Note that you can re-use config objects, e.g., if you want to create multiple similar datasets and the config is the same except for the respective ID. The following example shows how to insert one Activity dataset along with multiple ActivityLinks to various Projects.

MyTest_entity.testAction2b.onActionProcess

```
// IDs of the projects ot be linked to the Actitvity
var projectIds = ["c702e624-6675-4841-ac98-38da133a1c5b",
"559646cf-dcbf-4171-b251-952ac2ab9100",
"4436f590-1adb-466f-aad2-2cba0174aad7",
"9c78b5a2-36ee-45dd-9543-9099b78d28f2",
"029e0150-87bc-4f3a-9d34-7c455201f246"];

// config for creating the Activity dataset
var config = entities.createConfigForAddingRows();
config.entity("Activity_entity");

// config for creating ActivityLink datasets
var configLink = entities.createConfigForAddingRows();
configLink.fieldValues({
    "OBJECT_TYPE": "Salesproject"
}
);

// field values for creating the Activity dataset
config.fieldValues({
    "SUBJECT": subject,
    "TYPE": "LETTER",
    "ENTRYDATE": datetime.date().toString(),
    // link to configLink object
    "Links": [configLink]
}
);

// createRow is executed multiple times via a loop,
// each loop cycle with the same configLink object,
// but with different projectId values
for (let projectId of projectIds) {

    configLink.fieldValues({
        "OBJECT_ROWID": projectId
    }
    );

    entities.createRow(config);
}
```

Example 2:

Below you find an example code of a test Action that updates an existing Activity dataset.

MyTest_entity.testAction3a.onActionProcess

```
// creating the configuration object
var config = entities.createConfigForUpdatingRows();

// name of the Entity
config.entity("Activity_entity");

// mapping of the fields to be updated
config.fieldValues({
    "SUBJECT": "My new Subject value",
    "DIRECTION": "i"
});

// UID of the dataset to be updated
config.uid("38cb4fab-64f9-4d8e-aa6f-a158d13fc978");

// execution method for updating a dataset
entities.updateRow(config);
```

After executing this Action’s code, the "onDBUpdate" process of the given Entity will be executed.

Also for updates, you can re-use config objects, e.g., if you want to update multiple datasets, with the config being the same except for the respective ID. The following example shows how to set 3 different Persons (identified by their CONTACTID) inactive.

MyTest_entity.testAction3b.onActionProcess

```
// CONTACTIDs of Person datasets to be set inactive
var contactIdsToUpdate = ["4c9e95fe-25ae-4875-bd84-7b3705edd4fa",
    "27596cb7-2211-429b-801f-b428250496e8",
    "6263b12a-b19c-4870-97a4-1f044fe102e5"];

// one config for all changes
var config = entities.createConfigForUpdatingRows();
config.entity("Person_entity");
config.fieldValues({
    "STATUS": "CONTACTSTATINACTIVE"
});

// updateRow is executed multiple times via a loop,
// each loop cycle with the same config object,
// but with different CONTACTID values
for (let idToUpdate of contactIdsToUpdate) {
    config.uid(idToUpdate);
    entities.updateRow(config);
}
```

Example 3:

Below you find an example code of a test Action that deletes an existing Activity dataset.

MyTest_entity.testAction4.onActionProcess

```
// creating the configuration object
var config = entities.createConfigForDeletingRows();

// name of the Entity
config.entity("Activity_entity");

// UID of the dataset to be deleted
config.uid("38cb4fab-64f9-4d8e-aa6f-a158d13fc978");

// execution method for deleting a dataset
entities.deleteRow(config);
```

Before (!) executing this Action’s code, the "onDBDelete" process of the given Entity will be executed.

Example 4:

Below you find an example code of a test Action that creates an Activity dataset, along with ActivityLink datasets (linking the Activity to other Entities). As you can see, you can encapsulate multiple configurations with WriteEntity.

MyTest_entity.testAction5.onActionProcess

```
// encapsulated configuration for link1
var configLink1 = entities.createConfigForAddingRows();

// field mapping
configLink1.fieldValues({
    // "field" : "value"
    "OBJECT_TYPE": "Person",
    "OBJECT_ROWID": "c7ddf982-0e58-4152-b82b-8f5673b0b729"
});
```

```
// encapsulated configuration for link2
var configLink2 = entities.createConfigForAddingRows();

// field mapping
configLink2.fieldValues({
    "OBJECT_TYPE": "Organisation",
    "OBJECT_ROWID": "6efb4fab-64f9-4d8e-aa6f-a158d13fc273"
});


// now create a new Activity with ActivityLinks

// creating the configuration object
var config = entities.createConfigForAddingRows();

// name of the Entity
config.entity("Activity_entity");

//field mapping
config.fieldValues({
    "SUBJECT": "My Linked Activity",
    "INFO": "This is some demo information",
    "DIRECTION": "o",
    "ENTRYDATE": datetime.date().toString(),
    "CATEGORY": "MAIL",
    // connect the configurations
    // via Activity_entity's Consumer "Links"
    "Links": [configLink1, configLink2]
});

// execution method for creating a new dataset
var id = entities.createRow(config);

// loggin the automatically created UID of the new dataset,
// e.g., "88ae4fab-64f9-4d8e-aa6f-a158d13fd132"
logging.log("ACTIVITYID: " + id);
```

After executing this Action’s code, the onDBInsert process of the given Entity will be executed.

H.3. Usage in server processes

LoadEntity and WriteEntity can also be used in server processes. However, if you use it there, a user must be assigned. If required, simply create a "technical user" for that purposes, i.e., a user dataset that is not related to a real person but only to be used by specific internal logic.

H.4. Skipping prevalidation

Every of the Entity configs provides the `.skipPrevalidation(Boolean)` setter method. The default value within the config is `false`. In this default state the changes get validated before the Entity is saved. This prevents incomplete entries from being saved. If you set the value to true, validations are performed when saving data.

Using this method may be necessary when writing or changing at lot of data via processes as it reduces the number of validations and may lead to an increased performance.

> If you skip the prevalidation, you have to make sure your data is correct. Otherwise it may fail validation checks and incomplete data might be saved.

Appendix I: RecordContainerCache

In order to increase the performance of your ADITO system for repetitive requests of the same data, you can utilize a RecordContainerCache.

In ADITO, a cache is always defined *separately for each RecordContainer*. It is neither possible nor reasonable to cache simply "everything".

Only data generated by the RecordContainer can be cached, be it a dbRecordContainer (including "expression" properties), or a jDitoRecordContainer, with its contentProcess. On the contrary, e.g., the data generated or retrieved by the valueProcess of an EntityField cannot be cached (even if it interacts directly with the database!), as the valueProcess is not a part of the RecordContainer.

I.1. Basics

Generally, it makes sense to implement a RecordContainerCache, if

- data are relatively static (i.e., they do not change permanently)
- the amount of data is overseeable.

  Therefore, caching is only possible for RecordContainers that are **not pageable**.

- the same set of data is often requested, without any changes, and this is a problem for the system.
  - Example 1: The workload of the DBMS is unnecessary high, and the execution speed is slowed down. If data have not changed, it is faster to load them from a cache than from the database.
  - Example 2: ADITO is connected to an external, public, open source web service. If data have not changed, it is faster to load them from a cache than to utilize the web service.

Use cases in the xRM project are mostly helper lists, such as keywords, attributes, country-related data, languages, definitions of classifications, currency lists, price lists (if they change only once in a few months), district definitions, or other information related to any configurations.

Note that the usage of a cache itself consumes substantial system resources, particularly RAM. Therefore, a solid analysis of the users' behaviour (what data is actually requested repeatedly by whom, and how often is this the case?) and the amount of available RAM is required before deciding whether or not to utilize a cache for a specific RecordContainer.

`SELECT COUNT` queries are generally excluded from caching. Find more information in chapter COUNT queries.

I.2. Setup

By default, a RecordContainer does not use a cache. To activate caching, 2 propertys of the RecordContainer must be configured, which can be found in section "Cache" of the "Properties" window:

- cacheType
- cacheKeyProcess

As caching is not possible for pageable RecordContainers, these properties are only present, if property "isPageable" is set to false.

Furthermore, there is a project property named maxEntryLifetimeInCache, in order to limit the lifetime of a cache entry.

I.2.1. cacheType

The following cache types (scopes) are selectable:

- NONE: No caching. This is the default value for newly created RecordContainers.
- SESSION: This option is session-specific. One cache store is created separately for each user (assuming that each user opens only one session). The cache store for user A will be different from the cache store for user B. This is useful, if specific users often request specific data, differently from other users.
- GLOBAL: This option creates a common (shared) cache store for all sessions/users logged into the system. Example: If, for the first time, user A requests certain data, it will be loaded from the database. If, at a later time, user B requests exactly the same data, it will be loaded from the cache store instead of from the datbase - hence the loading process will be faster for user B and all other users requesting the same data.

In most cases, scope GLOBAL fits best to the users' requirements - also in case you utilize multiple languages (in this case, you only need to make sure that your cacheKey includes the locale).

I.2.2. cacheKeyProcess

In principle, a cache store is a list of key-value pairs, with the key being a unique identifier and the value being the set of requested data. Thus, the result of the cacheKeyProcess must be a unique key representing the requested data. If, e.g., 2 times the same set of data is requested, then exactly the same key must be generated. This enables caching: When a set of data is requested for the first time, it is loaded from the database and saved in the cache store, along with the unique key. When, at a later time, the same set of data is requested a further time, the RecordContainer first uses the same unique key to check the cache store for data associated with this key - and if it is found, it is loaded from there, not from the database.

The following factors influence the data generated by a RecordContainer and hence must be respected when constructing the cache key:

1. Components that determine, filter, and restrict data:
    - Lists of IDs to be included or excluded in the data query
    - Filters (user filters, search filters, permission filters, etc.)
    - Parameters evaluated in, e.g., the conditionProcess (dbRecordContainer) or rowCountProcess/contentProcess (jDitoRecordContainer).
2. Components that influence data presentation:
    - Language: Often relevant with RecordContainers, especially during translation of display values (e.g., Keywords).
    - Region: Can be significant in specific cases.
    - Sorting
    - Grouping

I.2.2.1. Helper functions

Although, in principle, you are free to construct the cache key as you like (as long as uniqueness is ensured and as long as the same request for a specific set of data always results in the same cache key), it is strongly recommended to utilize the specific helper functions provided by the ADITO platform:

The ADITO library "CachedRecordContainer_lib" (see, "process" > "libraries", in the project tree) already includes a helper class named `CachedRecordContainerUtils` that consists of several helper functions. These functions return a key string that can be used as result of the cacheKeyProcess. The helper functions read the values of various variables (lists of IDs, filter configurations, etc.) or Parameters and integrate these values into the key. Examples: `$local.idvalues`, `$local.filters`, or `$param.OnlyActives_param`.

If, at the time of the data request, a variable does not exist oder if it contains no value, its name is used instead of the value - which contributes to the requirement to make the key string unique.

All variable values/names are concatinated using a dot (".").

The following helper functions exist:

- `getKey` is the basic function. It enables you to define the complete key by yourself (via arbitrary variables as arguments) without the requirement to construct the string manually. In practice, `getKey` is seldomly used directly in a cacheKeyProcess; rather, it is internally called by the other helper functions (see below).
- `getKeyWithPreset` is used, if you want the cache key to respect all criteria that usually influences data, e.g., specific IDs, filters, sortings, and groupings. `getKeyWithPreset` internally calls function `getKey`, using the following arguments:
    - (mandatory:) a predefined set of variables (= "preset"), to be specified via a constant defined in class `CachedRecordContainerFieldPresets` (also part of CachedRecordContainer_lib). In particular, the following constants are available:
        - STANDARD: includes the variables `$local.idvalues`, `$local.idvaluesExcluded`, `$local.filters`, `$local.order`, and `$local.grouped`.
        - STANDARD_WITH_LOCALE: includes all variables of constant STANDARD plus (if present) the variable `$sys.clientlocale`.
    - (optionally:) an arbitrary number of additional variables
- `getCommonKey` internally calls function `getKeyWithPreset`, using the constant STANDARD_WITH_LOCALE (see above). Optionally, you can specify an arbitrary number of additional variables as arguments.

> These functions are well-documented: You can learn how to use them by reading their JSDoc. Furthermore, you can learn how they construct the cache key string, by inspecting their source code in the CachedRecordContainer_lib.

I.2.2.2. Examples in the xRM project

Examples of the design of a cacheKeyProcess can easily be found, if you simply search the complete xRM project for the string "CachedRecordContainerUtils.get".

Here is an example, used in the jDitoRecordContainer of Attribute_entity:

Attribute_entity.jDito.cacheKeyProcess

```
import { result } from "@aditosoftware/jdito-types";
import { CachedRecordContainerFieldPresets, CachedRecordContainerUtils } from "CachedRecordContainer_lib";

var key = CachedRecordContainerUtils.getCommonKey(
    "$param.AttributeCount_param",
    "$param.ChildId_param",
    "$param.ChildType_param",
    "$param.FilteredAttributeIds_param",
    "$param.GetOnlyFirstLevelChildren_param",
    "$param.IncludeParentRecord_param",
    "$param.ObjectType_param",
    "$param.ParentId_param",
    "$param.ParentType_param"
);
result.string(key);
```

Another example can be found in the dbRecordContainer of ResourcePlanning_entity:

ResourcePlanning_entity.db.cacheKeyProcess

```
import { CachedRecordContainerUtils } from "CachedRecordContainer_lib";
import { result } from "@aditosoftware/jdito-types";

var res = CachedRecordContainerUtils.getCommonKey(
    "$param.OrganisationContactIds_param",
    "$param.PersonContactIds_param",
    "$param.ResourceOperationIds_param"
);
result.string(res);
```

You can improve your understanding of the generation of the cache key by debugging or logging the results of the cacheKeyProcesses of various RecordContainers of the xRM project, in order to observe the generated key and its structure. Simply play around with, e.g., the filter in the web client, and see how the content of the key changes.
Furthermore, for testing purposes, you can also add (further) arguments to one of the helper functions (see above), e.g., new variables or Parameters. Keep in mind that the cache key will only change, if a variable/Parameter actually influences the SQL statement that retrieves the data.

I.2.2.3. Logged example

Let’s, for testing reasons, include a logging in the dbRecordContainer of KeywordEntry_entity:

KeywordEntry_entity.db.cacheKeyProcess

```
import { CachedRecordContainerFieldPresets, CachedRecordContainerUtils } from "CachedRecordContainer_lib";
import { logging, result } from "@aditosoftware/jdito-types";

var res = CachedRecordContainerUtils.getCommonKey(
    "$param.ContainerName_param",
    "$param.BlacklistIds_param",
     "$param.OnlyActives_param",
     "$param.WhitelistIds_param",
     "$param.Locale_param"
);

logging.log("------> Keyword Entry (db) Cache Key: " + res);

result.string(res);
```

Furthermore, make sure that the logging of database queries is active (see chapter Logging).

Now, open Context "Keyword Entry" and define, e.g., the filter "Keyword Category equal AddressType". Then apply the filter and watch the log. Among several other log entries, you should see

1. the key string generated by the cacheKeyProcess:

   ```
   (...) ------------> Keyword Entry (db) Cache Key: en_US._____$local.idvalues._____$local.idvaluesExcluded.{"type":"group",
   "operator":"AND","childs":[{"type":"row","name":"AB_KEYWORD_CATEGORY_ID","operator":"EQUAL","value":"AddressType","key":"1f70
   0fd2-5295-43a9-95ad-e73add4b5086","contenttype":"TEXT"}]}.{}._____$local.grouped._____$param.ContainerName_param._____$param
   .BlacklistIds_param.false._____$param.WhitelistIds_param._____$param.Locale_param
   ```

   → See how the key consists of a mixture of variable/Parameter values (e.g., the filter configuration) and variable/Parameter names (= names of variables/Parameters that do not exist or do not have a value). All variable/Parameter names/values are separated by a dot (".").

2. the database query used to load the filtered data:

   ```
   (...) SELECT AB_KEYWORD_ENTRY.TITLE , AB_KEYWORD_ENTRY.SORTING , AB_KEYWORD_ENTRY.ISESSENTIAL , AB_KEYWORD_ENTRY.ISACTIVE ,
   AB_KEYWORD_ENTRY.AB_KEYWORD_ENTRYID , AB_KEYWORD_ENTRY.KEYID , AB_KEYWORD_ENTRY.AB_KEYWORD_CATEGORY_ID , ( select
   AB_KEYWORD_CATEGORY.NAME from AB_KEYWORD_CATEGORY where AB_KEYWORD_CATEGORY.AB_KEYWORD_CATEGORYID = AB_KEYWORD_ENTRY
   .AB_KEYWORD_CATEGORY_ID )  AS CATEGORY_NAME   FROM AB_KEYWORD_ENTRY   WHERE AB_KEYWORD_ENTRY.AB_KEYWORD_CATEGORY_ID =
   '1f700fd2-5295-43a9-95ad-e73add4b5086'   ORDER BY CATEGORY_NAME , AB_KEYWORD_ENTRY.SORTING , AB_KEYWORD_ENTRY.TITLE ,
   AB_KEYWORD_ENTRY.AB_KEYWORD_ENTRYID
   ```

Subsequently, load the same data again, simply by clicking on the "refresh" button of your browser. Then, in the log, you can see the same key string again, but not the database query - which proofs that the cache is effective, as the repeatedly requested data has been loaded from the cache store, not from the database. Q.E.D.

I.2.3. Cache invalidation

In specific cases, it can be required to invalidate (= delete) the cache store (or parts of it) of a specific Entity. In particular, this is required, in order to avoid

- outdated cache store entries
- allocation of too much memory (RAM)

ADITO includes various automatisms and manual options in order to perform a cache invalidation, some of which refer to

- an individual cache store entry or
- the complete cache store of a specific RecordContainer or
- *all* cache stores of *all* RecordContainers of a project

I.2.3.1. Automatic

I.2.3.1.1. RecordContainer-specific

The ADITO platform includes an automatic cache invalidation, which is executed whenever data is changed (inserted, updated, or deleted) via a specific RecordContainer.

Example:
The dbRecordContainer of KeywordEntry_entity caches all requests of keyword entries. Now, when, e.g., the keyword entries of a specific keyword category have been cached (see the Logged example) and later the user adds a further keyword entry refering to the same keyword category, then the cache store of the dbRecordContainer of KeywordEntry_entity is outdated and must be refreshed - which is automatically initiated by the ADITO platform: The cache is deleted, in order to load the fresh data from the database (and cache it again), as soon as a user requests data of KeywordEntry_entity again. Thus, the cache store is now refreshed and provides the correct data for further requests.

I.2.3.1.2. Timespan-related

maxEntryLifetimeInCache is a project-related property (see project tree: preferences > PREFERENCES_PROJECT), which defines the amount of time an *individual* cache entry remains in any cache store of any Entity. Here, you can optionally change the default value to a value more suitable for your project. The property description explains the syntax, e.g., "1D 42M" means "1 day and 42 minutes".

This value needs to be set with great care:

The larger this time span is,

- the more data requests (cache store entries) are collected in the cache store, and thus the higher ("wider") is the effect of the cache; but
- the more memory (RAM) is required, and
- the higher is the probability that the user works with outdated data (in cases when, by mistake, there is neither an automatic nor a manual cache invalidation, see the other sub-chapters of this topic)

The smaller this time span is,

- the less data requests (cache store entries) are collected in the cache store, and thus the lower ("narrower") is the effect of the cache;
- the less memory (RAM) is required, and
- the lower is the probability that the user works with outdated data (in cases when, by mistake, there is neither an automatic nor a manual cache invalidation, see the other sub-chapters of this topic)

Therefore, the value of maxEntryLifetimeInCache needs to be set strictly according to the usual influencing factors, particularly

- the expected user behaviour, e.g.,
    - the expected frequency of data changes;
    - the expected kind and frequency of data requests;
- the memory (RAM) available for the ADITO system.

I.2.3.2. Manual

Besides the built-in automatic cache invalidation (see above), there are cases that require a manual cache invalidation. In particular, these are cases in which

1. data is changed (inserted, updated, deleted) *independent from the invalidation automatisms of the RecordContainer*. This happens, e.g., when the database is modified via direct SQL statements (using, e.g., `db.XXX` methods or the SqlBuilder), instead of using the RecordContainer-utilizing methods of "Write Entity" (see appendix chapter WriteEntity). A common use case is the inserting, updating, or deletion of data via an importer, which (for performance reasons) might use direct SQL statements.
2. the change of data of a specific Entity influences another (dependent) Entity. Example: If you add a new keyword category via the KeywordCategoryEdit_view, then the cached list of available keyword categories shown in the KeywordEntryEdit_view needs to be updated, in order to include also the new keyword category (see code example below).

In these kinds of cases, method `invalidateCache(`<name of Entity>, <name of RecordContainer>) must be executed. Here is an example from the ADITO xRM project:

KeywordCategory_entity.db.onDBInsert.js

```
import { entities } from "@aditosoftware/jdito-types";

//dependecies are updated so the cache needs to be updated
entities.invalidateCache("KeywordEntry_entity", "db");
```

If (in rare cases) it is required to invalidate *all* RecordContainerCaches of the complete project, simply execute method `invalidateCache()` without arguments. Here is an example from the ADITO xRM project, where a specific server process uses this method call:

mark_cachedrecordcontainers_invalid_serverProcess.process.js

```
import { entities } from "@aditosoftware/jdito-types";

entities.invalidateCache();
```

I.3. Shared caching with multiple ADITO servers

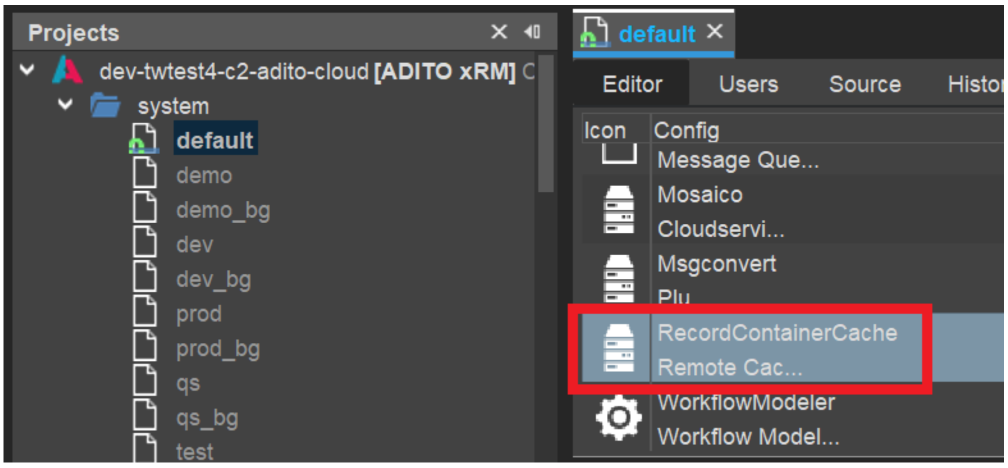
If your system includes multiple ADITO servers, it would be negative, if each server used only its own local cache store, independently from the cache stores of the other servers. In this case, server A had no information what happens on server B, and vice versa.

Rather, a shared (remote) cache must be applied, by utilizing a remote cache server. This server makes sure that all data requests in all sessions use one single (shared) cache store. (To be more precise: Internally, every server still has a local cache, called "NearCache", in order to reduce latency for accesses of the remote cache - but this architecture can be ignored here; it is enough to imagine the remote cache server as providing one single cache store, shared between all sessions of all servers.)

Example:
Given, in a multi-server environment, a cache of type GLOBAL has been configured for the RecordContainer of a specific Entity. User A logs in, which opens a session on, say, server A; user B logs in, which opens a session on, say, server B. Now, if user A requests a specific set of data of the respective Entity, then these data are loaded from the database and cached. Now, if every server used its own local cache and user B, subsequently, requests the same set of data, then the data would, again, be loaded from the database and cached a second time, this time on server B. If, however, the ADITO system utilized a remote cache server, then the data requested by user A would be stored in the shared cache store, and the (same) data requested by user B would be found and loaded from there - not again from the database.

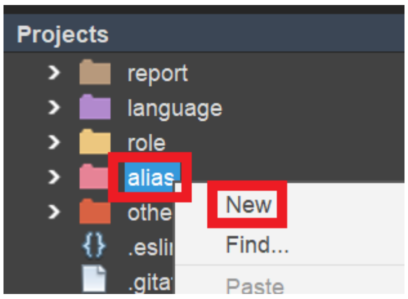
Using a shared cache only makes sense for caches of type GLOBAL. Caches of type SESSION are always restricted to one single session, and caches of other sessions are ignored, be they running on the same server or on other server - even if a remote cache server is active.

Therefore, each managed ADITO cloud system, by default, comes with an alias for a pre-configured, ready-to-use remote cache server. Its alias is named "RecordContainerCache" - see the AliasConfig (double-click on system > default, after the tunnel has been established):

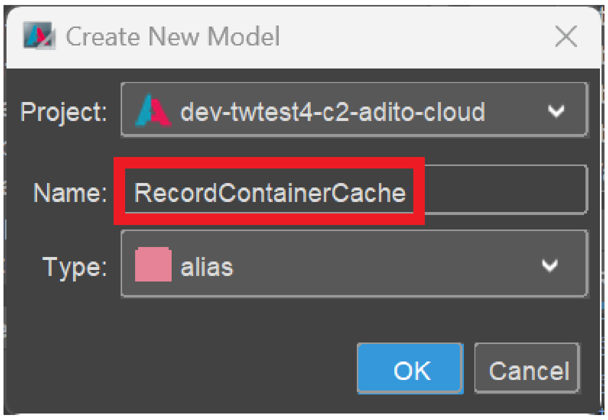


If this remote cache alias is not present yet, you need to add it first:

1. In the project tree, right-click on node "alias" and choose "New" from the context menu.



2. A dialog named "Create New Model" appears. Here, type in a suitable name (e.g., "RecordContainerCache").

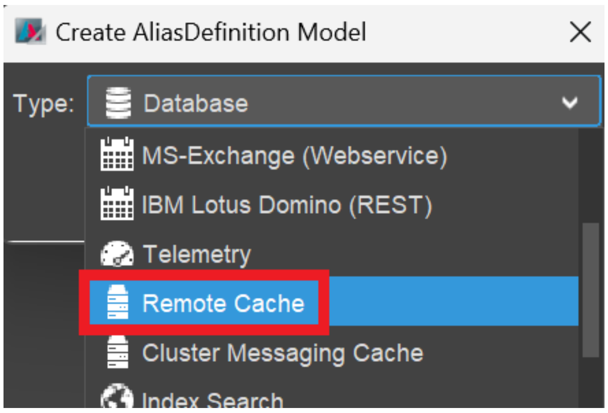

Create New Model

Project: dev-twtest4-c2-adito-cloud

Name: RecordContainerCache

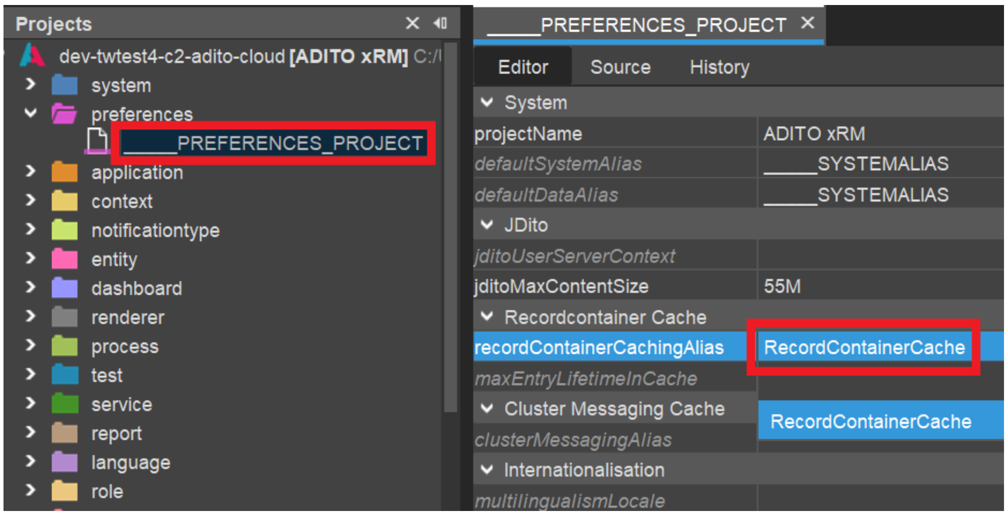
Type: alias

3. A dialog named "Create AliasDefinition Model" appears. Here, select the type "Remote Cache".



4. Deploy your project. Then, the new alias appears in the AliasConfig.

Now, check if the cache alias is set as value of the project property "recordContainerCachingAlias" (see preferences > ____PREFERENCES_PROJECT, in the project tree):

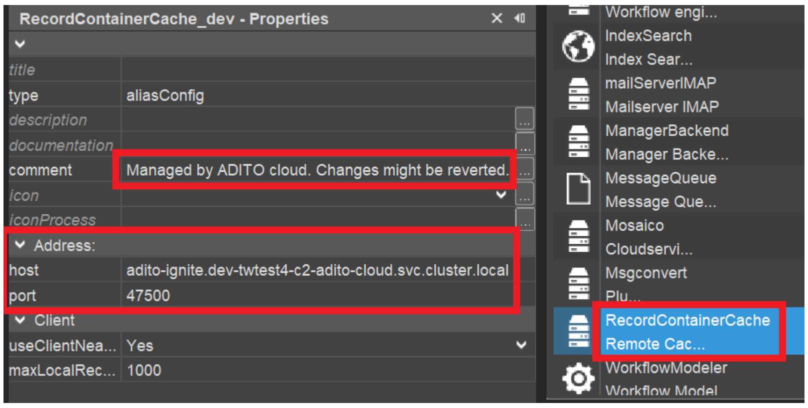


projectName	ADITO xRM
defaultSystemAlias	____SYSTEMALIAS
defaultDataAlias	____SYSTEMALIAS
jditoUserServerContext	
jditoMaxContentSize	55M
recordContainerCachingAlias	RecordContainerCache
maxEntryLifetimeInCache	
clusterMessagingAlias	
multilingualismLocale	

If it is not set yet,

1. set it now,
2. deploy your project,
3. restart the ADITO server,
4. re-establish the tunnel to your cloud system,
5. reconnect to your your system, in order to see the AliasConfig again.

Now, if you click on the cache alias in the AliasConfig, you can inspect its properties in the "Properties" window. Here, you should see that the address of the cache server (properties "host" and "port") has been set automatically:



RecordContainerCache_dev - Properties

title	
type	aliasConfig
description	
documentation	
comment	Managed by ADITO cloud. Changes might be reverted.
icon	
iconProcess	
Address:	
host	adito-ignite.dev-twtest4-c2-adito-cloud.svc.cluster.local
port	47500
Client	
useClientNea...	Yes
maxLocalRec...	1000

This semi-automatic activation of a remote cache server only works for managed ADITO cloud systems, as these systems by default come with a pre-configured, ready-to-use installation of a remote cache server. If, however, your system is an unmanaged cloud system, you first need to order the transformation of your system to a managed cloud system from ADITO.

ADITO does not offer support of integrating remote cache servers into "on premise" (not cloud-based) systems. Although, in principle, this is possible, the installation of the cache server and its integration as remote cache server must be realized by the customers themselves.

I.3.1. Alternative cache servers

By default, every managed ADITO cloud system comes with an installation of Apache Ignite as pre-configured, ready-to-use remote cache server. (Besides, ADITO utilizes Ignite also as cluster messaging server, see chapter Notifications with multiple ADITO servers.) In principle, you can also use other kinds of remote cache servers, but their installation and integration is not supported by ADITO (besides providing properties for the remote cache server’s host and port; see above).

Appendix J: EntityField/Keywords vs. Attributes

If you want to add flexible features to the datasets of an Entity (e.g., the color of a car, along with selectable values "green", "red", "blue", etc.) you have, in principle, at least the following 2 options:

1. Add an additional EntityField (e.g., CARCOLOR), which consumes KeywordEntries (e.g., GREEN, RED, BLUE), having a specific category (e.g. "CarColor").
2. Make the client-side setting of Attributes available, via an "Attribute" tab in the Entity’s MainView, whose content is filled via a Consumers connected with Attribute_entity’s corresponding Providers.
   You may study and copy this technique at the example of Organisation_entity’s View OrganisationAttributeRestriction_view (referencing AttributeRelationTree_view), whose content is filled via several Consumers named "AttributeXXX", along with several Parameters (in particular, ObjectRowId_param, and ObjectType_param). In the client, the result can be seen and used in the upper part of tab "Attributes" of Context "Company"'s MainView.
   Both approaches show advantages and disadvantages. Generally speaking, you should use Attributes only, if
   - their values are only used for displaying purposes, not for calculations;
   - or used only for a low number of datasets.

Otherwise you are likely to run into performance issues.

This means, an additional EntityField with Keywords should be preferred, if the values of this EntityField are not only to be displayed, but also used for evaluations, groupings, or complex filters.

In critical cases, it might be recommended to test the usability and performance of *both* approaches before deciding about what approach to use in the live system.

Find more details in the following chapters.

J.1. EntityField/Keywords

Advantages:

- higher performance when used for evaluations, groupings, complex filters, etc.
- better "visibility", if no value has been set yet: When using Attributes, no "suggestions" are shown, but you need to know in advance, what kind of feature can be assigned to a dataset - while an EntityField is always visible and can therefore suggest the user to examine its values and choose one.

- can be used in an index search (other than Attributes)
- can change its state (visible, mandatory, etc.), according to specific conditions (unlike Attributes)
- can be integrated in an ADITO Workflow
- The EntityField can be directly included in the access rights management (permissions); this means, e.g., that you can configure it in a way that it is only visibile or editable for user having a specific role.

Disadvantages:

- requires customizing - therefore, the effort for integrating and maintaining it, is higher
- if multiple Keyword-related EntityFields are used, the total number of EntityFields can get too high, and the Entity therefore confusing to understand and to maintain

Conclusion:
An additional EntityField with Keywords should be preferred, if the values of this EntityField are not only to be displayed, but also used for evaluations, groupings, complex filters, etc.

J.2. Attributes

Advantages:

- Once the general customizing is done (connection of the Entity with Attribute_entity, see above), no additional customizing in the Designer required, but everything else can be done in the client:
  - Arbitrary Attributes can be configured and maintained by the client administrator.
  - Those configured Attributes can then be set by the client user.
- See "Disadvantages" of previous chapter.

Disadvantages:

- Low performance is likely if used for evaluations, groupings, complex filters, etc.
- see "Advantages" of previous chapter

Conclusion:

Attributes should only be preferred, if their values are

- only used for displaying purposes, not for evaluations, groupings, complex filters, etc.
- or only used for a low number of datasets.

Appendix K: $sys variables

$sys variables are visible within one client and are independent from a specific context. They are typically used to store values that are used throughout the client, like global configurations, rights management via sales areas, etc. Find more information in appendix JDito system modules and variables.

Here is an overview of all $sys variables:

Name: $sys…	Description of return value
activeimage	internal ID of the active top image (entity, report, mail)
activewindow	internal ID of the active frame
ancestorimageuid	ID of all currently opened frames
calenderusers	calender users
clientcountry	country of the client
clientdata	DATA directory of the client. Visible with the variable $ADITO_DATA
clienthome	HOME directory of the client
clientid	ID of the client
clientlanguage	language of the client
clientlocale	localisation of the client (e.g. "de_DE")
clientos	name of operating system of the client
clienttemp	temp directory of the client
clientuid	UID of the client
clientvariant	language variant of the client
clientversion	version of the client (e.g 2021.0.1)

content	ID of the current selected record
currentcontextname	name of the current Context
currententityname	name of the current Entity
currentimage	ID of the current image variable
currentimagename	name of the current image variable
currentimagetype	type of the current image variable
datarow	condition of the current record without the keyword "where"
datarowcount	number of all datasets loaded in the RecordContainer (if so, respecting a filter), up to the limit configured in the RecordContainer (see property maximumDbRows)
datarowcountfull	number of all datasets (loaded or not) available via the RecordContainer (if so, respecting a filter), ignoring the limit configured in the RecordContainer (see property maximumDbRows)
date	system date as long
dbalias	active database alias
dynamicdate	system date as long
extendedpattern	current pattern as executed
filter	current selection of the frame. Object that includes: .filter, .permission, .filterCondition, .condition (e.g. vars.get("$sys.filter").filter)
filterable	boolean value stating whether or not the current View is filterable
groupable	boolean value stating whether or not the current View is groupable
groups	current active groups

hasSelection	boolean value stating whether or not it a selection is used
licenseid	ID of the license
operatingstate	current operatingstate of the Entity/View
order	order of the current records
origin	origin of the URL
outsidelineaccess	area code of the Provider (CTI)
pageable	boolean value stating whether or not the RecordContainer is pageable
parententity	name of the parent for the dependencies
parentuid	UID of the parent for dependencies
pattern	current pattern as entered
pendingpattern	returns the next pattern that will be executed
preferencesid	ID of the selected preferences if multiple choices are available
presentationmode	state of the current View/Entity
recordstate	state of the single record in the current View/Entity
scope	scope of the client
selection	current selection (for Entity or index)
selectionRows	all rows of a selection

selections	array as a multistring including the following information: - [0] Static: part of the selection that was used to open the Context/Entity/View - [1] Designer: part that was coded in the Designer - [2] search mask: part of search - [3] dependency: connection to the record - [4] extra: global search
serveraddress	address of the server (URL)
serverdata	DATA directory of the server
serverhome	HOME directory of the server
serverid	ID of the current system
serveros	operating system of the server
serverport	ports of the current server
servertemp	TEMP directory of the server
serverversion	version of the server
staticdate	current date as long
staticmillis	current date in milliseconds
superframe	name of the upper frame
superframeid	ID of the upper frame
superimage	name of the upper image
superimageid	ID of the upper image
superimagetype	type of the upper image
superwindowid	ID of the upper window

tablescanbecreated	boolean value stating whether or not records can be created in this Context
tablescanbedeleted	boolean value stating whether or not records can be deleted in this Context
tablescanbeedited	boolean value stating whether or not records can be edited in this Context
tableselection	condition for the seleceted table
tableviewselection	structured condition
timezone	timezone of the current user
today	current day without any time as long
uid	current UID of the record
uidcolumn	UID column name of the record
user	username of the current user
useridletime	idletime of the current user
usertoken	token of the current user
validationerrors	string with the current validation errors
viewmode	viewmode of the current component
workingmode	current workingmode
workingmodebefore save	workingmode that was active before a save action was triggered

Appendix L: $local variables

These system variables are only visible within specific processes and are mostly set by the ADITO core to pass values into these processes. Find more information in appendix JDito system modules and variables.

Here is a short list of selected $local variables (may be extended in future versions of this manual, according to demand).

This list does not include $local variables that are explained in other chapters of this manual. You will find them in the respective chapters, e.g. "$local.operator2" in chapter "FilterExtension", or "$local.value" in appendix "Accessing the value of an EntityField".

When using a full-text search for a $local variable over this document, consider that, for formatting reasons, some variable names include a line break, so you will find it not via their full name (e.g., "$local.initialRowdata"), but only via the second part of their name (e.g., "initialRowdata").

WARNING: Please do consider that this list is neither complete nor covering all cases. It contains only a few typical examples. Several $local variables have different meanings/purposes in different processes.

Name: $local...	Examples of return value	Data type	Examples of usage
condition	filter condition, e.g., "CONTACT.STATUS is null"	String	filterConditionProcess of a FilterExtension; groupQueryProcess of a FilterExtensionSet
filter	filter configuration (see below)	object	contentProcess of a jDitoRecordContainer
idvalue	IDs affected by a change in an indexRecordContainer	object	affectedIds process of a indexRecordContainer, see chapter "Index for 'Global Search'"

idvalues	ID affected by a change in a jDitoRecordContainer	String	contentProcess of a jDitoRecordContainer
initialRowdata	values of an Entity’s dataset at initial loading, i.e., BEFORE the user has made changes	object with EntityField names (as key) and their initial values	onDBUpdate process of a dbRecordContainer
lookupFieldName	(see below)	String	FilterExtensionSet for Attributes (see below)
rawvalue	user input (e.g., selection in combo box)	String	filterConditionProcess of a FilterExtension; find details in chapter "FilterExtension"
rowdata	values of an Entity’s dataset (if so) AFTER the user has made changes	object with EntityField names (as key) and their corresponding values	onDBdelete
uid	identifier of a dataset in a dbRecordContainer (see table in property linkInformation > column "Primary key" in line marked as "UID Table")	String	onDBInsert, onDBUpdate, onDBDelete

In the following, find detailed explanations of selected variables:

L.1. $local.filter

Example of structure and content of variable $local.filter

```
// if a filter is set, it looks like this:
{
    "filter": {
        "type": "group",
        "operator": "AND",
        "childs": [
            {
                "type": "row",
                "name": "ACTIVE",
                "operator": "EQUAL",
                "value": "Ja",
                "key": "true",
                "contenttype": "TEXT"
            }
        ]
    },
    "permissions": null,
    "subset": null,
    "ids": null
}

// if a filter is NOT set, it looks like this:
{
    "filter": null,
    "permissions": null,
    "subset": null,
    "ids": null
}
```

L.2. $local.lookupFieldName

- Availability:
  This variable is available in the valueProcess of a Parameter of a Consumer.
- Content:
  - If the Consumer is connected to an EntityField (→ lookup), then the variable contains the name of this EntityField.
  - FilterExtensions and FilterExtensionSets can also use a Consumer for a lookup. In these cases, the variable contains the name of the FilterExtensionField.
- Use case in xRM: FilterExtensionSet for Attributes: With some FilterExtensionFields, a Consumer is used for the values. To inform the Consumer, which FilterExtensionField has been selected, the name of the field is retrieved via $local.lookupFieldName.

Appendix M: $property variables

Name: $property.MY_FIELD...	Examples of return value	Data type	Examples of usage
contentDescription		String	current contentDescription
contentTitle		String	current contentTitle
contentType		String	current contentType
dropDown	key + value	Object	result of a dropDownProcess, without it being calculated anew. Via the key, the displayed value can be resolved.
iconId		String	current iconId
image			current image
metadata			current metadata
state			current state
title		String	current title
titlePlural		String	current titlePlural

Appendix N: XML in JDito

This appendix is about working with XML in JDito. XML is short for e*X*tensible *M*arkup *L*anguage, which is a way to express various data in a structured form. XML consists of HTML-like tags and their attributes, but those tags and attributes are not predefined, but rather defined by the developer. XML is often used in APIs or webservices to exchange data in a defined format.

If you have the choice between using XML and JSON, you should always choose JSON as it is native to JavaScript and JDito and it’s more lightweight in terms of memory usage. Only use XML if an external API requires it and JSON is not an option.

In JDito you have access to the XML and XMLList object, which can be imported like `import { XML } from "@aditosoftware/jdito-types";`. This object offers methods for building the XML script in a builder-like way. It can also take a XML script as a string at instantiation to prefill the object based on that. XMLList is used to represent an XML document containing multiple elements and XML is used to represent one element.

Simple example:

```
import { XML, logging } from "@aditosoftware/jdito-types";

var xmlObject = new XML("<xml> \
                            <element1 attribute1='value1'
attribute2='value2'>  \
                                element1 content \
                            </element1> \
                        <xml>");

logging.log("Element1 content: " + xmlObject.element1 + "\nElement1
attribute1: " + xmlObject.element1["attribute1"]);
```

To access the elements, the typical object notation is used. If you’re dealing with simple XML scripts, this is the preferred way. More complex XML scripts may require the usage of the XML object’s methods like `.child()`, `.children()` or `.appendChild()` to build or process the XML. As an XML element can contain further elements or even contain several elements having the same name, `.child()` returns an array of XML objects.

Example of reading an XML containing multiple children of the same name:

```
import { XML, logging } from "@aditosoftware/jdito-types";
//initializing as object from a XML string
```

```
var xmlObject = new XML("<xml> \
                            <element1 attribute1='value1'
attribute2='value2'>  \
                                element1 content \
                            </element1> \
                            <element1 attribute1='value1'
attribute2='value2'>  \
                                element1 content \
                            </element1> \
                            <element1 attribute1='value1'
attribute2='value2'>  \
                                element1 content \
                            </element1> \
                        <xml>");

//iterating over the children of the XML object
for(let child in xmlObject.children())
{
    logging.log("Content: " + child.text() + "\nAttribute 1: " +
child["attribute1"] + "\nAttribute 2: " + child["attribute2"] );
}
```

Example of building a XML script:

```
import { XMLList, XML, logging } from "@aditosoftware/jdito-types";

// Task: A JSON object containing our data needs to be
// "converted" into an XML. (Names are chosen generically.)

var data = {
    {
        "attribute1":"value1", "attribute2":"value2", "content":
"element content"
    }
    ,{
        "attribute1":"value1", "attribute2":"value2", "content":
"element content"
    }
    ,{
        "attribute1":"value1", "attribute2":"value2", "content":
"element content"
    }
};

// preparing the main element of our XML

var xmlListObj = new XMLList("<data></data>");
```

```
// appending children based on the content of your data JSON

for(let obj in data)
{
    xmlListObj.appendChild(new XML("<element1 attribute1='" + obj
.attribute1 + "' attribute2='" + obj.attribute2 + "'> " + obj
.content + "</element1>"));
}

//checking the generated XML in the server log

logging.log(xmlListObj.toXMLString());
```

Do no longer handle XML using the inline syntax (E4X), example:

```
var myXml = <element1>
                <element2>My first text.</element2>
                <element3>My second text.</element3>
            </element1>;
```

Appendix O: Car pool example: EntityFields

As the spelling of the EntityFields' names is essential for the function of the following code examples, you can find the names of all car pool related EntityFields below, ready for "copy & paste".

Furthermore, the following tables indicate the preferable contentType of the EntityField, as well as the data type of the corresponding database column.

Table 12. Car_entity

Name	contentType	Liquibase data type
availability	BOOLEAN	- (calc.)
CARID	TEXT	CHAR(36)
COLOR	TEXT	VARCHAR(36)
CURRENCY	TEXT	VARCHAR(36)
damages	TEXT	- (calc.)
LICENSEPLATENUMBER	TEXT	NVARCHAR(20)
MANUFACTUREDATE	DATE	DATE
MANUFACTURER	TEXT	VARCHAR(36)
mileage	NUMBER	- (calc.)
PICTURE	IMAGE	LONGBLOB
PRICE	NUMBER	DECIMAL(10,2)
TYPE	TEXT	NVARCHAR(30)
carValue	NUMBER	- (calc.)

Table 13. CarDriver_entity

Name	contentType	Liquibase data type
age	NUMBER	- (calc.)
CARDRIVERID	TEXT	CHAR(36)
CONTACT_ID	TEXT	CHAR(36)
drivingExperience	NUMBER	- (calc.)
DRIVINGLICENSENUMBER	TEXT	NVARCHAR(30)
DRIVINGLICENSEISSUEDATE	DATE	DATE
parkingTicketFinesSum	NUMBER	- (calc.)
speedingFinesSum	NUMBER	- (calc.)

Table 14. CarReservation_entity

Name	contentType	Liquibase data type
CAR_ID	TEXT	CHAR(36)
CARDRIVER_ID	TEXT	CHAR(36)
CARRESERVATIONID	TEXT	CHAR(36)
CURRENCY	TEXT	VARCHAR(36)
DAMAGE	TEXT	NVARCHAR(300)
ENDDATE	DATE	DATETIME
MILEAGERETURN	NUMBER	INT
mileageStart	NUMBER	- (calc.)
PARKINGTICKETFINE	NUMBER	DECIMAL(7,2)

Name	contentType	Liquibase data type
SPEEDINGFINE	NUMBER	DECIMAL(7,2)
STARTDATE	DATE	DATETIME

Appendix P: ResourceTimeline example: Liquibase and code

P.1. Liquibase

A folder named resourceTimelineExample was created in the top level of the Data_alias alias definition. You can adopt this or change it to your own chosen path.

Changelog:

```
<?xml version="1.1" encoding="UTF-8" standalone="no"?>
<databaseChangeLog xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.6.xsd">
    <include relativeToChangelogFile="true" file="resTimeline_creates.xml"/>
</databaseChangeLog>
```

Changeset:

```
<?xml version="1.1" encoding="UTF-8" standalone="no"?>
<databaseChangeLog xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.6.xsd">
  <changeSet author="" id="8e3865a8-6824-4ff2-a5da-cd4351e674c7">
    <createTable tableName="EXAMPLERESOURCE">
            <column name="BUSINESSHOURFROM" type="NVARCHAR(10)">
                <constraints nullable="false"/>
            </column>
            <column name="BUSINESSHOURTO" type="NVARCHAR(10)">
                <constraints nullable="false"/>
            </column>
            <column name="CONTACT_ID" type="VARCHAR(36)">
                <constraints nullable="false"/>
            </column>
            <column name="EXAMPLERESOURCEID" type="CHAR(36)">
                <constraints primaryKey="true" primaryKeyName="PK_EXAMPLERESOURCE_EXAMPLERESOURCEID"/>
            </column>
            <column name="DATE_NEW" type="DATETIME"/>
            <column name="USER_NEW" type="VARCHAR(36)"/>
            <column name="DATE_EDIT" type="DATETIME"/>
            <column name="USER_EDIT" type="VARCHAR(36)"/>
    </createTable>
    <createTable tableName="EXAMPLEPLANNINGENTRY">
            <column name="EXAMPLEOPERATION_ID" type="VARCHAR(36)">
                <constraints nullable="false"/>
            </column>
            <column name="EXAMPLERESOURCE_ID" type="VARCHAR(36)">
                <constraints nullable="false"/>
            </column>
            <column name="DATE_START" type="DATETIME">
                <constraints nullable="false"/>
            </column>
            <column name="DATE_END" type="DATETIME">
                <constraints nullable="false"/>
            </column>
            <column name="EXAMPLEPLANNINGENTRYID" type="CHAR(36)">
                <constraints primaryKey="true" primaryKeyName="PK_EXAMPLEPLANNINGENTRY_EXAMPLEPLANNINGENTRYID"/>
            </column>
            <column name="DATE_NEW" type="DATETIME"/>
            <column name="USER_NEW" type="VARCHAR(36)"/>
            <column name="DATE_EDIT" type="DATETIME"/>
            <column name="USER_EDIT" type="VARCHAR(36)"/>
    </createTable>
    <createTable tableName="EXAMPLEOPERATION">
            <column name="TITLE" type="NVARCHAR(512)">
                <constraints nullable="false"/>
            </column>
            <column name="INFO" type="NCLOB"/>
            <column name="EXAMPLEOPERATIONID" type="CHAR(36)">
                <constraints primaryKey="true" primaryKeyName="PK_EXAMPLEOPERATION_EXAMPLEOPERATIONID"/>
            </column>
            <column name="DATE_NEW" type="DATETIME"/>
            <column name="USER_NEW" type="VARCHAR(36)"/>
            <column name="DATE_EDIT" type="DATETIME"/>
            <column name="USER_EDIT" type="VARCHAR(36)"/>
        </createTable>
    </changeSet>
</databaseChangeLog>
```

P.2. Code

Code for DATE_NEW fields: valueProcess

```
import { result, neon, vars } from "@aditosoftware/jdito-types";

if(vars.get("$sys.recordstate") == neon.OPERATINGSTATE_NEW && !vars.get("$this.value"))
{
    result.string(vars.get("$sys.date"));
}
```

Code for DATE_EDIT fields: valueProcess

```
import { result, neon, vars } from "@aditosoftware/jdito-types";

if(vars.get("$sys.recordstate") == neon.OPERATINGSTATE_EDIT && !vars.get("$this.value"))
{
    result.string(vars.get("$sys.date"));
}
```

Code for USER_NEW fields: valueProcess

```
import { result, neon, vars } from "@aditosoftware/jdito-types";

if(vars.get("$sys.recordstate") == neon.OPERATINGSTATE_NEW && !vars.get("$this.value"))
{
    result.string(vars.get("$sys.user"));
}
```

Code for USER_EDIT fields: valueProcess

```
import { result, neon, vars } from "@aditosoftware/jdito-types";

if(vars.get("$sys.recordstate") == neon.OPERATINGSTATE_EDIT && !vars.get("$this.value"))
{
    result.string(vars.get("$sys.user"));
}
```

Code for EXAMPLERESOURCEID, EXAMPLEPLANNINGENTRYID and EXAMPLEOPERATIONID fields; valueProcess

```
import { util, vars, result, neon } from "@aditosoftware/jdito-types";

if(vars.get("$sys.recordstate") == neon.OPERATINGSTATE_NEW && !vars.get("this.value"))
{
    result.string(util.getNewUUID());
}
```

Appendix Q: Content types

For every EntityField, you need to select one of multiple content types, via property contentType. Some of the content types have special options via additional properties, e.g., to limit the size of their input. If the content is set via property contentTypeProcess, all additional properties are displayed; but still only those of them are evaluated that belong to the respective content type.

Here is a list of available content types:

Table 15. Available values for property contentType

Content type	Description
TEXT	Sequence of strings, with the option to limit the size via property maxFieldSize
LONG_TEXT	Similar to TEXT, but with the option to distribute the text to multiple lines (press ENTER for new line)
NUMBER	Numbers, with the option to limit the size via properties minValue, maxValue, maxIntegerDigits, and maxFractionDigits. Via properties outputFormat and inputFormat, a fixed format can be determined (not recommended for international environments). There is an in-built validation, if the input is actually a number: If you enter, e.g., letters or special characters, the save button is disabled, and the message "Wrong number format" is shown.
DATE	Date values, with the option to set the accuracy via property "resolution" (e.g., MONTH, DAY, or HOUR), which determines the function of the "Date picker". Via properties outputFormat and inputFormat, a fixed format can be determined (not recommended for international environments) - but be aware that still property "resolution" determines the accuracy of saving the value.
HTML	Allows input via a HTML editor, which saves an HTML string. You can limit the input size via property maxFieldSize. Via property "htmlEditorFeatures" you can determine if the editor should have only basic features, like font stylings, or also advanced features, like tables.

Content type	Description
IMAGE	Provides a component to upload an image, either via a file chooser or via drag & drop. Can also show icons (VAADIN:*, NEON:*) or the colored placeholder icons (TEXT:*). If the user chooses a file that is no image (.png, .jpg, etc.), no error message is shown, but the component remains in default state, and nothing is saved. The data type of the corresponding database colum must be selected accordingly, e.g., LONGBLOB for MariaDB.
TELEPHONE	Provides an input field for a telephone number. After saving, the content is shown with a hyperlink that leads to a computer telephone integration (cti). PLEASE NOTE: There is no validation, if the input has a valid telephone number format. This has to be done additionaly, via an onValidation process (see, e.g., EntityField PHONE_ADDRESS of Employee_entity).
EMAIL	Provides an input field for an email address. After saving, the content is shown with a hyperlink that leads to the standard email client of the user. PLEASE NOTE: There is no validation, if the input has a valid email format. This has to be done additionaly, via an onValidation process.
LINK	Provides an input field for a hyperlink. After saving, the content is shown with a hyperlink that leads to the standard browser of the user. PLEASE NOTE: There is no validation, if the input has a valid hyperlink format. This has to be done additionaly, via an onValidation process.
PASSWORD	Provides an input field for a password. During input, one asterisk is shown instead of each character, and after saving, three asterisks are always shown, independently from the actual length of the password. PLEASE NOTE: In the database, the password is still saved in cleartext.
SIGNATURE	Provides an input field for a signature, to be written on a rectangular field with the mouse pointer or with another suitable input device. The signature is saved as base64 string (data:image/png;base64). The data type in the database should be seleced accordingly, e.g., LONGTEXT for MariaDB.
FILE	Provides a file upload component that works both via a file browser and via drag & drop. The file is saved as base64 string (data:image/png;base64). The data type in the database should be seleced accordingly, e.g., LONGTEXT for MariaDB.

Content type	Description
FILESIZE	Provides an input field for entering the size of data, in Bytes. After saving, the number will automatically be shown with a suitable unit, e.g., Byte, kB, MB, or GB.
BOOLEAN	Provides a slider component for entering a boolean value. The slider on the left position means "false", the slider on the right positioin means "true". In the ADITO client, the corresponding values are, by default, shown as "No" (meaning "false") or "Yes" (meaning "true"), whatever data type you may use for the corresponding database column. Works with various data types, e.g., BOOLEAN, CHAR, VARCHAR, INT, etc. Exception: If you set a decimal data type (e.g., DECIMAL(5,2)), the client shows "0.00" instead of "No" ("false"), and "1.00" instead of "Yes" ("true").
FILTER_TREE	Provides a button "Open extended filter conditions", which opens a popup window, in which you can define extended filter conditions - just like the ones you know from the filter component of a FilterView. Requires at least an empty filter to be preset, e.g., via the valueProcess - see, e.g., EntityField FILTER of TopicTreeTopicConfiguration_entity. Otherwise you get an error message, explaining "The value of the node 'filter' is expected."

Appendix R: Siblings vs. refreshParent

The Entity parameter "siblings" and the Consumer parameter "refreshParent" both reflect the situation that there are dependencies between 2 Entities.

Now, when to use Siblings and when refreshParent?

- Siblings are suitable for 2 Entities that are relatively similar, and you want to keep them updated
- refreshParent is the standard to be used for updating normal connections (e.g., parent-child)

Example:
If you work with multiple tabs that, e.g., all show tickets, then "siblings" refreshes all these tabs and thus increases the server workload. Furthermore, the timer for the closing of the session is set back, which increases the memory usage. When using refreshParent, however, only the effected tab/dataset is updated, which thus is more performant.

Appendix S: LexoRank

If you are not interested in basics and background information, and you simply want to know how to integrate LexoRank in an ADITO Context, you may skip the following chapters and continue reading with chapter Example implementation.

S.1. Introduction

LexoRank is an algorithm used as core of a list ordering system. It originates in the software Jira, developed by company Atlassian.

With LexoRank ("Lexo" stands for "Lexicographic"), an order of elements is established not by only numbers (1 < 2 < 3 < …9564739), but in an alphanumerical way, e.g.,

- a < b
- a < aa
- aa < ab
- aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa < aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab
- a1 < a2
- a1a < a1b
- etc.

This leads to lists that can be (re-)ordered arbitrarily by drag-and-drop, with the content of the sorting fields being generated automatically, e.g.,

- a
- b
- c
- ci
- cii
- cj
- d
- …

S.2. Benefits

The benefits of LexoRank are, amongst others,

1. to solve the performance problem that you can have with numerical ordering systems.
2. to simplify drag and drop.

S.2.1. Performance

Example:
Given a table with a numerical ordering system, using integers, e.g.:

Order	Name
1	Cat
2	Dog
3	Bird
4	Horse
5	Pig

If you now, e.g., want to drag "Cat" from the first position and drop it down to the position between "Horse" and "Pig" you have the problem that there is no integer between 4 and 5. Therefore, you have to re-index ("rebalance") almost all order fields:

Order	Name
1	Dog
2	Bird
3	Horse
4	Cat
5	Pig

While the rebalancing process is running, any changes (insert, delete, drag and drop) must be completely disabled.

If you have very long tables, this requirement of frequent rebalancings can lead to substantial performance issues. Instead, with LexoRank, you only need to calculate the order field value of the moved row.

Example:
Given a table that uses LexoRank as (alphanumerical) ordering system, e.g.:

Order	Name
a	Cat
b	Dog
c	Bird
d	Horse
e	Pig

If you now, e.g., want to drag "Cat" from the first position and drop it down to the position between "Horse" and "Pig", the LexoRank algorithm only needs to calculate a suitable order field value for the moved row, e.g., "di", while all other order field values remain unchanged:

Order	Name
b	Dog
c	Bird
d	Horse
di	Cat
e	Pig

As you can see here, the problem that there is no letter between "d" and "e" is solved by introducing a further digit and setting its value, e.g., to "di" (because d < di < e). This is not possible with numerical ordering systems, as (in the above example), e.g., "45" is not a value between "4" and "5", but 4 < 5 < 45. (Allowing decimal numbers like "4.5" would only shift the problem, as, in practice, decimal numbers usually have very limited decimal places.)

As you can see, the more digits you allow for your ordering field, the more powerful your ordering system will be.

> Still, in practice, you will have a parameter defining the maximum number of digits. If this maximum is reached, a rebalancing will also be required, like it is for numerical ordering systems (see above). In ADITO xRM, by default, rebalancing is not integrated. Rather, the database field holding the order value usually is of data type VARCHAR(255), allowing 255 digits. Thus, the probability of running out of digits is close to zero. (And, if the rare case of reaching the maximum should really happen, drag-and-drop would not be disabled for the whole table, but only for cases that require the generation of an ordering value that exceeds the maximum.)

S.2.2. Drag and drop

Formerly, in ADITO, enabling drag and drop required multiple additional fields, like ITEMSORT, SORTINGLAYER, and SORTINGVALUE. Now, only one field is required, holding the LexoRank value (in ADITO named LEXORANK). Thus, also the related code can be simplified significantly.

S.3. Further information

In this manual, LexoRank will not be explained in detail, as an internet search for the term will lead you to various web sites with further information, e.g.,

- https://confluence.atlassian.com/adminjiraserver/managing-lexorank-938847803.html
- https://tmcalm.nl/blog/lexorank-jira-ranking-system-explained/
- https://lexorank.richardboeh.me/
- https://medium.com/whisperarts/lexorank-what-are-they-and-how-to-use-them-for-efficient-list-sorting-a48fc4e7849f
- https://medium.com/turkcell/lexorank-managing-sorted-tables-with-ease-f404f7eb00a9

S.4. Usage in ADITO

There are several reference implementations inspired by LexoRank, with lexorank4j being the wrapper used in ADITO, particularly simplifying drag and drop in ViewTemplates of type TreeTable. Although, in the strict sense, the term "LexoRank" is restricted to the algorithm included in Jira, it is used in the broader sense in this documentation - meaning any algorithm that follows the same principles as LexoRank in Jira.

LexoRank was introduced in ADITO mainly as ordering system that simplifies drag and drop in TreeTables (e.g., in the "Offeritem" Context). Besides, performance could be optimized, e.g., in the

"TopicTree" Context. The LexoRank functionality is mainly provided via the library "lexorank" of the ADITO platform (core). The main function is the calculation of the value of an ordering field (i.e., an additional EntityField, usually named "LEXORANK"), in the following cases:

1. A row is added in the TreeTable. (This also works for other tables, like MultiEditTables, and for Trees.)
2. A row is moved in the TreeTable, via drag and drop.

Although the "lexorank" library includes powerful functionality, you still need some customizing, in order to add LexoRank to your ADITO project (see chapter Example implementation).

S.4.1. Format

The format of the LexoRank values is a combination of

- the "bucket" number (0, 1, or 2 - see below)
- a 6-digit alphanumerical string, separated by a vertical bar, and followed by a colon
- optional further digits

Examples:

- "0|abc123:"
- "0|abc123:xy5".

Its maximum length is defined via the data type of the corresponding database column LEXORANK (default: VARCHAR(255)) and via the property "maxLexoRankLength" of the corresponding Provider.

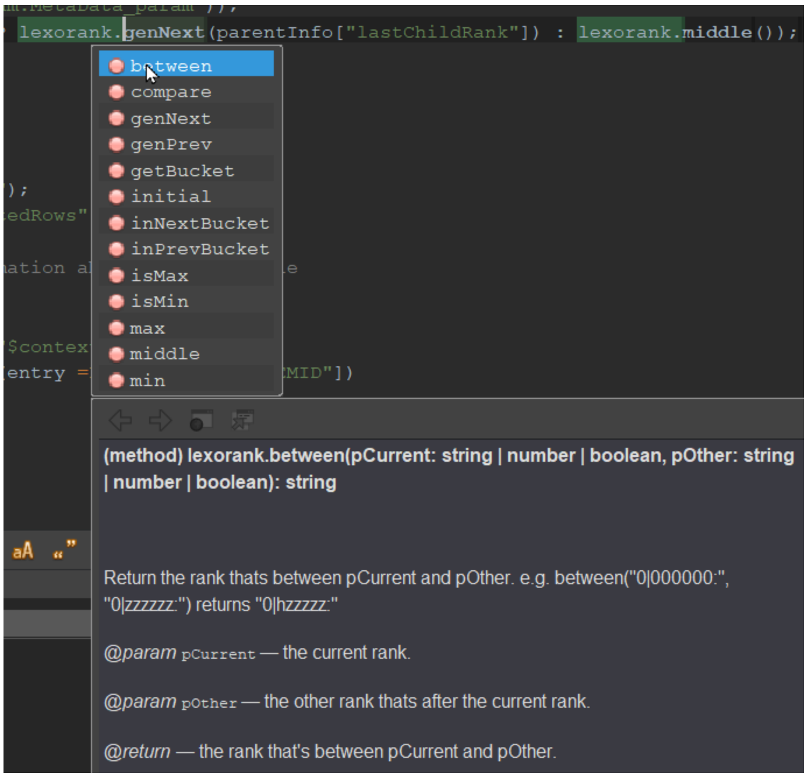
"Buckets" are used in the context of rebalancing. In the internet you can find various explanations of LexoRank format, including the purpose of "buckets" - see, e.g., here.

S.4.2. Mainly used methods

In the customizing of ADITO applications, the following methods of library lexorank are mainly used:

- `lexorank.genNext(<current rank>)`: Generates the next rank from the provided current rank. e.g. `genNext("0|000000:")` returns `0|100000:`
- `lexorank.middle()`: Returns the rank that is in between the min and the max rank.

An overview of the names and JSDoc of *all* methods of library lexorank is available, as usual, via the autocompletion:


```
lexorank.genNext(parentInfo["lastChildRank"]) : lexorank.middle();
```

(method) lexorank.between(pCurrent: string | number | boolean, pOther: string | number | boolean): string

Return the rank thats between pCurrent and pOther. e.g. between("0|000000:", "0|zzzzzz:") returns "0|hzzzzz:"

@param `pCurrent` — the current rank.

@param `pOther` — the other rank thats after the current rank.

@return — the rank that's between pCurrent and pOther.

S.4.3. Rebalancing

The implementation of LexoRank in ADITO does not include rebalancing (see the corresponding note in chapter Performance). However, on demand, rebalancing can be added according to the LexoRank algorithm in Jira, as described, e.g., here. To enable this, the format of the LexoRank value, in ADITO, already includes a preceding bucket number (see chapter Format).

S.5. Example implementation

In the following sub-chapters, the integration of LexoRank in ADITO will be explained using the example of Context "Offeritem". Here, LexoRank is used for supporting the Views OfferitemFilter_view (with ViewTemplate OfferitemsTreeTable) and OfferitemMultiEdit_view (with ViewTemplate OfferitemsMultiEditTable). In the web client, you can find both Views subordinated to OfferOfferitem_view, which in turn is a part of OfferMain_view.

The main purpose of introducing LexoRank here was not primarily performance, but the simplification of drag and drop (see chapter Benefits with sub-chapter Drag and drop).

The naming has no technical reason. Nevertheless, you should keep to the naming conventions as given below, e.g., to name the central EntityField LEXORANK. This will ensure consistency to existing LexoRank implementations and simplify orientation for other developers.

S.5.1. Introduce new database column LEXORANK

The first step, when integrating LexoRank, is to introduce a new database column named LEXORANK, with data type VARCHAR(255). (255 is the current default size/length for LEXORANK in ADITO. This value can be changed on demand. Anyway, it must match the property maxLexoRankLength of the related Provider - see sub-chapter Set sorting properties.)

You may use the following Liquibase snipped in your changelog:

```
<changeSet author="j.smith" id="256ddeb8-292f-456d-99b6-9e75d5305ab5">
  <addColumn tableName="OFFERITEM">
    <column name="LEXORANK" type="VARCHAR(255)"/>
  </addColumn>
</changeSet>
```

Depending on whether there already is another ordering system in your application (e.g., based on fields like ITEMSORT etc.), you may include a suitable "sql" tag in your Liquibase changeset, in order to enable a smooth transfer to LexoRank, keeping the current order. Example:

```
<changeSet author="j.smith" id="256ddeb8-292f-456d-99b6-9e75d5305ab5">
(...)
  <sql> update OFFERITEM set LEXORANK = CONCAT ( CONCAT ( CONCAT ( CONCAT ( CONCAT ( '0|i', floor(ITEMSORT / 1000) % 10 ), floor
(ITEMSORT / 100) % 10 ), floor(ITEMSORT / 10) % 10 ), ITEMSORT % 10 ), '0:' ) </sql>
</changeSet>
```

S.5.2. Introduce new EntityField LEXORANK

The next (and main) step is to introduce and configure a new EntityField named LEXORANK (here: in Offeritem_entity), leaving all properties in default state (contentType remains TEXT), except for

- title: LexoRank
- valueProcess: see below
- RecordFieldMapping: Connect the EntityField LEXORANK with the corresponding database column LEXORANK, as usual.

S.5.3. Set valueProcess

Set the valueProcess of EntityField LEXORANK as follows:

```
if(neon.OPERATINGSTATE_NEW == vars.get("$sys.recordstate") && Utils.isNullOrEmptyString("$this.value"))
{
    let lastLexorankUnderParent = newSelect("MAX(OFFERITEM.LEXORANK)")
        .from("OFFERITEM")
        .where("OFFERITEM.ASSIGNEDTO", vars.get("$field.ASSIGNEDTO"))
        cell();
    result.string(lexorank.genNext(lastLexorankUnderParent));
}
```

The above valueProcess code is simple and universal to use. In the original Offeritem_entity, the valueProcess of EntityField LEXORANK uses a Parameter named MetaData_param, which retrieves additional information about the parent item. As this is not required for LexoRank itself, we can ignore it here.

S.5.4. Set sorting properties

In the Provider that is consumed in the Context holding the superordinated View (in our example, this is Offeritem_entity's Provider "OfferItems") set the following properties:

- sortingField: LEXORANK
- sortingMethod: LEXORANK
- maxLexoRankLength: 255 (This is the current default value in ADITO. It can be changed on demand. Anyway, it must match the size/length given in the data type of the corresponding database column LEXORANK, see sub-chapter Introduce new database column LEXORANK.)

Only a TreeTable should access this Provider. As the ViewTemplate MultiEditTable does not support it, Offeritem_entity has 2 Providers - one for TreeTable and one for MultiEditTable. (Otherwise, MultiEditTable would override the LexoRank with a simple sorting 1,2,3…).

S.5.5. enableDragAndDrop

In the TreeTableViewTemplate, make sure property "enableDragAndDrop" is set to true.

S.6. Further examples

Besides Offeritem, you can find further LexoRank implementation examples in ADITO xRM, e.g., in the following Contexts:

- Orderitem: see OrderMain_view > OrderOrderitem_view > OrderitemFilter_view > Treetable
- TopicTree: see TopicTreeFilter_view > treeTable
- ResourceOperationTask (from ADITO version 2024.2.2):

- see ResourceOperationTaskTemplateMain_view > ResourceOperationTaskFilterAddFromTemplateAction_view > TreeTable
- see ResourceOperationMain_view > ResourceOperationTask_view > ResourceOperationTaskFilterPlannedDrawer_view > TreeTable

Appendix T: Trainee example

In the following you can find the files you need to prepare the "Trainee" example that is used to explain the configuration of a FilterExtensionSet.

T.1. Extending the changelog.xml files

In the project tree, under alias > Data_alias, create a new folder named "trainee". Then create the following new changelog file in the folder "trainee".

alias/Data_alias/trainee/changelog.xml

```
<?xml version="1.1" encoding="UTF-8" standalone="no"?>
<databaseChangeLog xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.6.xsd">
    <include relativeToChangelogFile="true" file="create_trainee.xml"/>
</databaseChangeLog>
```

Then add a reference to the new changelog.xml file in the "master" changelog.xml file:

alias/Data_alias/changelog.xml

```
<?xml version="1.1" encoding="UTF-8" standalone="no"?>
<databaseChangeLog xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
        xmlns:ext="http://www.liquibase.org/xml/ns/dbchangelog-ext"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog-ext http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-
ext.xsd http://www.liquibase.org/xml/ns/dbchangelog http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-4.4.xsd">
    <include relativeToChangelogFile="true" file="../Loghistory_alias/changelog.xml"/>

(...)

    <include relativeToChangelogFile="true" file="../Demodata_Data_alias/changelog.xml"/>
        <include relativeToChangelogFile="true" file="trainee/changelog.xml"/>
</databaseChangeLog>
```

T.2. Creating the database table

In the "trainee" folder, add the following file and name it "create_trainee.xml".

alias/Data_alias/trainee/create_trainee.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<databaseChangeLog xmlns="http://www.liquibase.org/xml/ns/dbchangelog" xmlns:ext="http://www.liquibase.org/xml/ns/dbchangelog-ext"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog-ext
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-ext.xsd http://www.liquibase.org/xml/ns/dbchangelog
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-4.1.xsd">
  <changeSet author="p.dietl" id="a4d54535-7193-4148-b6b7-58624b5e05fd">
    <preConditions onFail="MARK_RAN">
      <not>
        <tableExists tableName="trainee" />
      </not>
    </preConditions>
    <createTable tableName="trainee">
      <column name="TRAINEEID" type="CHAR(36)">
        <constraints nullable="false" primaryKey="true" />
      </column>
      <column name="FIRSTNAME" type="VARCHAR(30)" />
      <column name="LASTNAME" type="VARCHAR(30)" />
      <column defaultValueComputed="NULL" name="BIRTHDAY" type="date" />
```

```
        <column name="GENDER" type="VARCHAR(36)" />
        <column defaultValueComputed="NULL" name="GRADEENGLISH" type="INT" />
        <column defaultValueComputed="NULL" name="GRADEGERMAN" type="INT" />
        <column defaultValueComputed="NULL" name="GRADEMATH" type="INT" />
        <column defaultValueComputed="NULL" name="PICTURE" type="LONGBLOB" />
        <column name="TYEAR" type="VARCHAR(50)" />
      </createTable>
    </changeSet>
    <changeSet author="p.dietl" id="a4d54535-7193-4148-b6b7-58624b5e05f7">
      <insert tableName="trainee">
        <column name="TRAINEEID" value="17c57879-31f0-4ec3-b510-8efa414b6127" />
        <column name="FIRSTNAME" value="John" />
        <column name="LASTNAME" value="Smith" />
        <column name="BIRTHDAY" valueDate="1996-05-23" />
        <column name="GENDER" value="m" />
        <column name="GRADEGERMAN" valueNumeric="3" />
        <column name="GRADEMATH" valueNumeric="2" />
        <column name="TYEAR" value="1" />
      </insert>
      <insert tableName="trainee">
        <column name="TRAINEEID" value="19588327-6191-4a63-be40-0ea617690f0f" />
        <column name="FIRSTNAME" value="Luke" />
        <column name="LASTNAME" value="Taylor" />
        <column name="BIRTHDAY" valueDate="2009-01-26" />
        <column name="GENDER" value="m" />
        <column name="GRADEGERMAN" valueNumeric="4" />
        <column name="GRADEMATH" valueNumeric="4" />
        <column name="PICTURE" />
        <column name="TYEAR" value="2" />
      </insert>
      <insert tableName="trainee">
        <column name="TRAINEEID" value="633d69a4-a64b-4356-a870-b55fb1cef10b" />
        <column name="FIRSTNAME" value="Anne" />
        <column name="LASTNAME" value="Miller" />
        <column name="BIRTHDAY" valueDate="2010-04-13" />
        <column name="GENDER" value="f" />
        <column name="GRADEENGLISH" valueNumeric="2" />
          <column name="GRADEMATH" valueNumeric="4" />
        <column name="TYEAR" value="3" />
      </insert>
      <insert tableName="trainee">
        <column name="TRAINEEID" value="6539126d-4a59-413e-a468-4bc36b5ae7f5" />
        <column name="FIRSTNAME" value="Thomas" />
        <column name="LASTNAME" value="Hiller" />
        <column name="BIRTHDAY" valueDate="2008-10-28" />
        <column name="GENDER" value="m" />
        <column name="GRADEENGLISH" valueNumeric="2" />
        <column name="GRADEMATH" valueNumeric="1" />
        <column name="PICTURE" />
        <column name="TYEAR" value="2" />
      </insert>
    </changeSet>
  </databaseChangeLog>
```

T.3. Executing a Liquibase update

Execute a Liquibase update on the "master" changelog.xml. After a few seconds, you should be able to see the new database table "trainee" in the database editor of the ADITO Designer.

T.4. Creating the Entity

In the project tree, navigate to folder "entity" and create a new Entity named "Trainee_entity". Open the "Source" tab of the new Entity and replace its content by the following .aod code:

entity/Trainee_entity

```
<?xml version="1.0" encoding="UTF-8"?>
<entity xmlns="http://www.adito.de/2018/ao/Model" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" VERSION="1.4.0"
xsi:schemaLocation="http://www.adito.de/2018/ao/Model adito://models/xsd/entity/1.4.0">
  <name>Trainee_entity</name>
  <majorModelMode>DISTRIBUTED</majorModelMode>
```

```xml
    <icon>VAADIN:GAMEPAD</icon>
    <contentTitleProcess>%aditoprj%/entity/Trainee_entity/contentTitleProcess.js</contentTitleProcess>
    <iconId>VAADIN:ABACUS</iconId>
    <recordContainer>db</recordContainer>
    <entityFields>
      <entityProvider>
        <name>#PROVIDER</name>
      </entityProvider>
      <entityProvider>
        <name>#PROVIDER_AGGREGATES</name>
        <useAggregates v="true" />
      </entityProvider>
      <entityField>
        <name>TRAINEEID</name>
        <title>Traineeid</title>
        <groupable v="false" />
      </entityField>
      <entityField>
        <name>GENDER</name>
        <title>Gender</title>
        <groupable v="false" />
        <dropDownProcess>%aditoprj%/entity/Trainee_entity/entityfields/gender/dropDownProcess.js</dropDownProcess>
      </entityField>
      <entityField>
        <name>FIRSTNAME</name>
        <title>Firstname</title>
        <groupable v="false" />
        <mandatory v="true" />
      </entityField>
      <entityField>
        <name>LASTNAME</name>
        <title>Lastname</title>
        <groupable v="false" />
        <mandatory v="true" />
      </entityField>
      <entityField>
        <name>BIRTHDAY</name>
        <title>Birthday</title>
        <contentType>DATE</contentType>
        <resolution>DAY</resolution>
        <groupable v="false" />
      </entityField>
      <entityField>
        <name>GRADEENGLISH</name>
        <title>English</title>
        <colorProcess>%aditoprj%/entity/Trainee_entity/entityfields/gradeenglish/colorProcess.js</colorProcess>
        <contentType>NUMBER</contentType>
        <maxIntegerDigits v="1" />
        <maxFractionDigits v="2" />
        <groupable v="false" />
      </entityField>
      <entityField>
        <name>GRADEGERMAN</name>
        <title>German</title>
        <colorProcess>%aditoprj%/entity/Trainee_entity/entityfields/gradegerman/colorProcess.js</colorProcess>
        <contentType>NUMBER</contentType>
        <groupable v="false" />
      </entityField>
      <entityField>
        <name>GRADEMATH</name>
        <title>Math</title>
        <colorProcess>%aditoprj%/entity/Trainee_entity/entityfields/grademath/colorProcess.js</colorProcess>
        <contentType>NUMBER</contentType>
        <groupable v="false" />
      </entityField>
      <entityField>
        <name>fullName</name>
        <title>Full Name</title>
        <groupable v="false" />
        <valueProcess>%aditoprj%/entity/Trainee_entity/entityfields/fullname/valueProcess.js</valueProcess>
      </entityField>
      <entityField>
        <name>gradeAverage</name>
        <title>Average Grade</title>
        <colorProcess>%aditoprj%/entity/Trainee_entity/entityfields/gradeaverage/colorProcess.js</colorProcess>
        <contentType>NUMBER</contentType>
        <groupable v="false" />
        <valueProcess>%aditoprj%/entity/Trainee_entity/entityfields/gradeaverage/valueProcess.js</valueProcess>
      </entityField>
      <entityActionField>
        <name>showOverallAverage</name>
        <title>Show Overall Average</title>
        <onActionProcess>%aditoprj%/entity/Trainee_entity/entityfields/showoverallaverage/onActionProcess.js</onActionProcess>
        <iconId>VAADIN:ABACUS</iconId>
      </entityActionField>
      <entityField>
        <name>icon</name>
        <contentType>IMAGE</contentType>
        <groupable v="false" />
        <valueProcess>%aditoprj%/entity/Trainee_entity/entityfields/icon/valueProcess.js</valueProcess>
      </entityField>
      <entityField>
        <name>age</name>
        <title>Age</title>
        <groupable v="false" />
        <valueProcess>%aditoprj%/entity/Trainee_entity/entityfields/age/valueProcess.js</valueProcess>
      </entityField>
      <entityField>
        <name>PICTURE</name>
        <title>Picture</title>
        <contentType>IMAGE</contentType>
        <groupable v="false" />
        <displayValueProcess>%aditoprj%/entity/Trainee_entity/entityfields/picture/displayValueProcess.js</displayValueProcess>
      </entityField>
      <entityField>
        <name>TYEAR</name>
        <title>Year of training</title>
        <groupable v="false" />
        <dropDownProcess>%aditoprj%/entity/Trainee_entity/entityfields/tyear/dropDownProcess.js</dropDownProcess>
      </entityField>
      <entityConsumer>
        <name>KeywordGenders</name>
        <dependency>
          <name>dependency</name>
          <entityName>KeywordEntry_entity</entityName>
          <fieldName>SpecificContainerKeywords</fieldName>
        </dependency>
        <children>
          <entityParameter>
            <name>ContainerName_param</name>
            <valueProcess>
%aditoprj%/entity/Trainee_entity/entityfields/keywordgenders/children/containername_param/valueProcess.js</valueProcess>
            <expose v="false" />
          </entityParameter>
        </children>
      </entityConsumer>
    </entityFields>
    <recordContainers>
      <dbRecordContainer>
        <name>db</name>
        <alias>Data_alias</alias>
        <recordFieldMappings>
          <dbRecordFieldMapping>
            <name>BIRTHDAY.value</name>
            <recordfield>TRAINEE.BIRTHDAY</recordfield>
            <isFilterable v="true" />
          </dbRecordFieldMapping>
          <dbRecordFieldMapping>
            <name>FIRSTNAME.value</name>
            <recordfield>TRAINEE.FIRSTNAME</recordfield>
            <isFilterable v="true" />
          </dbRecordFieldMapping>
          <dbRecordFieldMapping>
            <name>GENDER.value</name>
            <recordfield>TRAINEE.GENDER</recordfield>
            <isFilterable v="true" />
          </dbRecordFieldMapping>
          <dbRecordFieldMapping>
            <name>GRADEENGLISH.value</name>
            <recordfield>TRAINEE.GRADEENGLISH</recordfield>
            <isFilterable v="true" />
          </dbRecordFieldMapping>
          <dbRecordFieldMapping>
            <name>GRADEGERMAN.value</name>
            <recordfield>TRAINEE.GRADEGERMAN</recordfield>
            <isFilterable v="true" />
          </dbRecordFieldMapping>
          <dbRecordFieldMapping>
            <name>GRADEMATH.value</name>
            <recordfield>TRAINEE.GRADEMATH</recordfield>
            <isFilterable v="true" />
          </dbRecordFieldMapping>
          <dbRecordFieldMapping>
            <name>LASTNAME.value</name>
            <recordfield>TRAINEE.LASTNAME</recordfield>
            <isFilterable v="true" />
          </dbRecordFieldMapping>
          <dbRecordFieldMapping>
            <name>TRAINEEID.value</name>
            <recordfield>TRAINEE.TRAINEEID</recordfield>
            <isFilterable v="true" />
          </dbRecordFieldMapping>
          <dbRecordFieldMapping>
            <name>PICTURE.value</name>
            <recordfield>TRAINEE.PICTURE</recordfield>
          </dbRecordFieldMapping>
          <dbRecordFieldMapping>
            <name>TYEAR.value</name>
            <recordfield>TRAINEE.TYEAR</recordfield>
          </dbRecordFieldMapping>
          <dbRecordFieldMapping>
            <name>fullName.value</name>
            <expression>
  %aditoprj%/entity/Trainee_entity/recordcontainers/db/recordfieldmappings/fullname.value/expression.js</expression>
          </dbRecordFieldMapping>
        </recordFieldMappings>
        <linkInformation>
          <linkInformation>
            <name>TRAINEE</name>
            <tableName>TRAINEE</tableName>
            <primaryKey>TRAINEEID</primaryKey>
            <isUIDTable v="true" />
            <readonly v="false" />
          </linkInformation>
        </linkInformation>
        <filterExtensions>
          <filterExtensionSet>
            <name>example_filterSet</name>

  <filterFieldsProcess>%aditoprj%/entity/Trainee_entity/recordcontainers/db/filterextensions/example_filterset/filterFieldsProcess.js</f
  ilterFieldsProcess>

  <filterValuesProcess>%aditoprj%/entity/Trainee_entity/recordcontainers/db/filterextensions/example_filterset/filterValuesProcess.js</f
  ilterValuesProcess>

  <filterConditionProcess>%aditoprj%/entity/Trainee_entity/recordcontainers/db/filterextensions/example_filterset/filterConditionProcess
  .js</filterConditionProcess>
            <isGroupable v="true" />

  <groupQueryProcess>%aditoprj%/entity/Trainee_entity/recordcontainers/db/filterextensions/example_filterset/groupQueryProcess.js</group
  QueryProcess>
            <filtertype>BASIC</filtertype>
          </filterExtensionSet>
        </filterExtensions>
      </dbRecordContainer>
    </recordContainers>
  </entity>
```

T.5. Creating Context and FilterView

In the project tree, navigate to folder "context" and create a new Context named "Trainee". Set the new Context’s property "entity" to "Trainee_entity".

Create a new View for the "Trainee" context and name it TraineeFilter_view. Open the new View’s tab "Source" and replace its content by the following .aod code:

context/Trainee/TraineeFilter_view

```
<?xml version="1.0" encoding="UTF-8"?>
<neonView xmlns="http://www.adito.de/2018/ao/Model" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" VERSION="1.2.3"
xsi:schemaLocation="http://www.adito.de/2018/ao/Model adito://models/xsd/neonView/1.2.3">
  <name>TraineeFilter_view</name>
  <majorModelMode>DISTRIBUTED</majorModelMode>
  <filterable v="true" />
  <layout>
    <groupLayout />
  </layout>
  <children>
    <tableViewTemplate>
      <name>table</name>
      <columns>
```

```
        <neonTableColumn>
          <name>c37bea5c-c392-4d15-9fe6-cb78fde71f44</name>
          <entityField>PICTURE</entityField>
        </neonTableColumn>
        <neonTableColumn>
          <name>304b0639-465d-4bc3-99af-cbeae503061f</name>
          <entityField>icon</entityField>
        </neonTableColumn>
        <neonTableColumn>
          <name>693f9d93-a5af-469c-92d2-40bf803d4335</name>
          <entityField>fullName</entityField>
        </neonTableColumn>
        <neonTableColumn>
          <name>11bc9624-2e5e-46fa-aa0c-c589e1e828be</name>
          <entityField>BIRTHDAY</entityField>
        </neonTableColumn>
        <neonTableColumn>
          <name>29846123-c312-4ec8-9a1d-b60b6084254a</name>
          <entityField>age</entityField>
        </neonTableColumn>
        <neonTableColumn>
          <name>4eae11fc-78e6-449b-b78e-0261b4085921</name>
          <entityField>GENDER</entityField>
        </neonTableColumn>
        <neonTableColumn>
          <name>81beca91-5f97-44f6-b081-2a6bb913ab6d</name>
          <entityField>TYEAR</entityField>
        </neonTableColumn>
        <neonTableColumn>
          <name>a99c5447-ef47-4d5e-bd77-6346fbceeee6</name>
          <entityField>GRADEENGLISH</entityField>
        </neonTableColumn>
        <neonTableColumn>
          <name>e78f0f90-b82c-455e-85be-091fdeb46290</name>
          <entityField>GRADEGERMAN</entityField>
        </neonTableColumn>
        <neonTableColumn>
          <name>b152e8f8-aa70-4b56-9084-d86418227bde</name>
          <entityField>GRADEMATH</entityField>
        </neonTableColumn>
        <neonTableColumn>
          <name>70908d27-330e-4574-986e-c645466627ad</name>
          <entityField>gradeAverage</entityField>
        </neonTableColumn>
      </columns>
    </tableViewTemplate>
    <treeTableViewTemplate>
      <name>treetable</name>
      <columns>
        <neonTreeTableColumn>
          <name>ad9d3a2c-d14a-4337-86b0-7dfdc9dbc319</name>
          <entityField>PICTURE</entityField>
        </neonTreeTableColumn>
        <neonTreeTableColumn>
          <name>4e9dad95-ecc0-4bdd-8d4e-43c73ddc3680</name>
          <entityField>icon</entityField>
        </neonTreeTableColumn>
        <neonTreeTableColumn>
          <name>22b9cc6d-df6b-4028-9571-c3791f813d31</name>
          <entityField>fullName</entityField>
        </neonTreeTableColumn>
        <neonTreeTableColumn>
          <name>a857676b-b6ed-4bdf-aaec-c354176831a7</name>
          <entityField>BIRTHDAY</entityField>
        </neonTreeTableColumn>
        <neonTreeTableColumn>
          <name>6763a5cd-7fa3-4ee7-93b6-0dcfaf8fee94</name>
          <entityField>age</entityField>
        </neonTreeTableColumn>
        <neonTreeTableColumn>
          <name>091e3e63-fe3c-4855-bd47-c3efe97323dd</name>
          <entityField>GENDER</entityField>
        </neonTreeTableColumn>
        <neonTreeTableColumn>
          <name>b7356bb8-d671-4755-84e7-41b4196af971</name>
          <entityField>TYEAR</entityField>
        </neonTreeTableColumn>
        <neonTreeTableColumn>
          <name>d033c989-85a3-4346-b2ff-c9fc465c852d</name>
          <entityField>GRADEENGLISH</entityField>
        </neonTreeTableColumn>
        <neonTreeTableColumn>
          <name>c06f06cb-8f8a-4804-a315-d060f61af8d7</name>
          <entityField>GRADEGERMAN</entityField>
        </neonTreeTableColumn>
        <neonTreeTableColumn>
          <name>f9c9e994-ceed-4451-bb55-2209b974bedc</name>
          <entityField>GRADEMATH</entityField>
        </neonTreeTableColumn>
        <neonTreeTableColumn>
          <name>9ad174dd-6913-47df-9981-7f2ec05e6342</name>
          <entityField>gradeAverage</entityField>
        </neonTreeTableColumn>
      </columns>
    </treeTableViewTemplate>
  </children>
</neonView>
```

Now, set the "Trainee" Context’s property "filterView" to "TraineeFilter_view".

T.6. Adding the Context to the Global Menu

In the project tree, open the menu editor (application > _____SYSTEM_APPLICATION_NEON) and add the new Context "trainee" to the Global Menu. (You may create a new menu group for it, titled, e.g., "Trainee Management".)

Deploy, logout, and login to the web client. Open the new Context "Trainee". Now, you can continue with chapter FilterExtensionSet.

Appendix U: Version history

Version	Changes
2024.1.3	• New chapter Context filter (content search), describing the behavior of the content search bar, which is available in various ViewTemplates. • Chapter FilterExtensionSet improved by example that is easier to understand. • Chapter IndexRecordContainer reduced to a short introduction, followed by a reference to AID093 Indexsearch, where all required information is included now. • Chapter Add Dashlets improved by 2 screenshots showing configuration of property "fragment". • Appendix LoadEntity and WriteEntity: New chapter getRow vs. getRows • Appendix System variables improved by additional explanations to $global variables and other system variable types. • Bugfix in chapter Adding an ATTRIBUTES tab: Attribute_lib → AttributeUtil_lib • Various minor optimizations.
2024.1.2	• New chapter Adding Tasks. • New sub-chapter on Renderer MULTISELECTCOMBOBOX. • New sub-chapter Export of a subordinated Entity. • New sub-chapter on variable $local.lookupFieldName. • New appendix LexoRank, with reference in chapter TreeTable. • Chapter Structure of ADITO projects: Added info box about modularization. • Improved description of automatisms in the Database Recordcontainer, e.g., at the end of chapter Connecting EntityFields with database columns (RecordContainer). • Improved description of adding an Observation. • Various minor optimizations.
2024.1.1	• Chapter ResourceTimeline: New sub-chapter Specific color constants. • Minor grammatical optimizations.

Version	Changes
2024.1	• New chapter on GridLayout • New chapter on $local.rowdata and $local.initialRowdata in RecordContainer-related processes • Chapter JDitoRecordContainer: Added prompts not to use `$field` variables in onInsert/onUpdate/onDelete processes. • Chapter WriteEntity: Setter method `.fieldValues`: Added info box on nonexistent validation against value lists. • Various bugfixes, updates, rearrangements, and optimizations. • This is the official version of the Customizing Manual for ADITO version 2024.1.
2024.0	• Updated chapter RecordContainers, including new introduction and new chapter COUNT queries • New appendix RecordContainerCache with extensive information about how to utilize a cache in ADITO, including explanations regarding shared caching with Apache Ignite (see chapter Shared caching with multiple ADITO servers) • New chapter Notifications and observations, including explanations regarding distributed notification management with Apache Ignite (see chapter Notifications with multiple ADITO servers) • Included references to the new ADITO Information Document AID121 "Themes" in chapter Themes and in appendix Requirements for customized Theme. • New appendix Version history, including summaries of changes in previous versions of this document. This appendix replaces the previous long table at the beginning of the manual, which had forced the reader to scroll down a lot of pages before reaching the table of content. • Updated chapter Export (required entry in Dependency_lib) • Minor bugfixes • Minor linguistical refactoring • This is the official version of the Customizing Manual for ADITO version 2024.0.

Version	Changes
2023.1.1	• Some refactoring to various code examples • Added a reference to an example of a multi selection action to chapter Multi Selection Action
2023.1	• Chapter Assigning layout and ViewTemplates: Added description of how to configure the content search bar (Context filter) • New chapter Export • New appendix Siblings vs. refreshParent • Chapter Customized logging: Added "Tip" box recommending the optimal parameter setting of `JSON.stringify` • minor improvements and bugfixes • This is the official version of the Customizing Manual for ADITO version 2023.1.

Version	Changes
2023.0	• New chapter $this.value and $field.MYFIELD in valueProcess • New chapter Visibility of tabs • New chapter Field Groups • Chapter JDitoRecordContainer: ◦ Bugfix in example code of property onUpdate ◦ Step-by-step example added ◦ Chapter Filtering a JDitoRecordContainer added • New chapter Lookup for translated values • Chapter Add Dashlets: Explanation of mandatory icon added • Chapter Adding a LOGS tab extended by explanation of further useful custom properties • Chapter LoadEntity and WriteEntity: Warning box extended with respect to performance • New chapter Storing user-specific data outside ASYS_USERS • New appendix Content types • New chapter Paging • New chapter Creating new project roles • Chapter Retrieving pending records extended and improved • Chapter about ViewTemplate Gantt extended, incl. description of property "isSubstep" • Chapter EntityRecordsRecipe, sub-chapter Usage in customized methods: Added note box regarding usage of method `.filter` • Chapter Configuring EntityFields: ◦ added warning regarding outputFormat ◦ added description of length-restricting properties • Chapter FilterExtension: Included not regarding index • Chapters FilterExtension and FilterExtensionSet: Extended explanations of groupQueryProcess

Version	Changes
2022.2	• Chapter Adding a LOGS tab: Added a screenshot detailing where to find the Auditmode of a table • Code examples using SqlBuilder got updated to a more recent syntax. (newSelect) • Corrected several typos and bugs in code examples • Code example Person_entity.db.age.value.expression rebuilt using Sql helper functions • Added section Skipping prevalidation to Appendix Load/Write Entity • changed "isSelectionAction" to new "selectionType" in Actions and ActionGroups • added description of `expandRootItems` property to chapters Tree and TreeTable • added an info box to FilterExtension detailing the use of FilterExtensions and FilterExtensionSets on RecordContainers without paging • added detailed description and a simplified implementation example of the new ResourceTimeline ViewTemplate ResourceTimeline • This is the official version of the Customizing Manual for ADITO version 2022.2.
2022.1	• Chapter Creating Entities: Warning box about not to overlook the refactoring tab • Info box about one [userDirectory] being generated for every ADITO version (major/minor/hotfix). • New chapter Adding an ATTRIBUTES tab • New chapter Troubleshooting > Bug tracking • New appendix about using XML in JDito • Appendix chapter WriteEntity ○ Bugfix in code examples ○ Deprecated "for each" in code examples replaced by standard syntax "for…of" (ES2015) • minor improvements and bugfixes

Version	Changes
2022.0	• New chapter Adding a LOGS tab • New chapter Using gif files • Extended description of RecordFieldMapping’s property [expression], including an info box regarding its invalidity, if property "recordField" is set • chapter What is JDito?: new warning box that system-reserved names must not be used for variable naming • chapter Configure Dashboard defaults updated regarding new Dashboard editor • chapter How does a "$field" variable get its value? extended by description of criteria for the automatic loading/calculation of a field • appendix LoadEntity: Description of effect if an empty Array is passed to setter method `uids` • chapter EntityRecordsRecipe: Description of effect if an empty Array is passed to setter method `uidsIncludelist` • chapter JDitoRecordContainer: EntityField named "UID" must always be present • new chapter Avatars • chapter Themes: Description of how to view the available colors • chapter Color: Effect on Avatars included • chapter Internationalization: new sub-chapters User help and Validation of address and communication data • new chapter Device-specific designs • new chapter Retrieving pending records • chapter 360Degree Context updated and extended by referencing property "documentation" of 360Degree_entity. • bugfix in code of init_car.xml (values of MANUFACTUREDATE) - see chapter Entering example data • appendix WriteEntity extended by info box explaining the importance of EntityField order when using method `.fieldValues` • new chapter explaining ViewTemplate Favorite

Version	Changes
2021.2	• new chapter EntityRecordsRecipe • new chapter Dynamic filter values • new chapter FilterBuilder; FilterBuilder included in code fragments (instead of JSON) • new chapter Prerequisites, explaining the prerequisites for reading this manual, including a description of how to activate an extended Logging of database access and JDito processes • chapter Trigger, explaining the principles when the valueProcess of an EntityField is being executed. • chapter Configuring EntityFields: Reference to AID066 regarding max. resolution/size of images • chapter JDitoRecordContainer: Improved description of property [hasDependentRecords] • appendix Order of execution of Entity processes: Description of variable "$local.modifiertype" in paragraph about property [onValueChange] • chapter Calculated fields restructured and enhanced by remark regarding "checking for null" in valueProcess, in order to avoid unintended overwriting of existing values • chapter Translate all extended by notes on settings required for DeepL API. • reference to AID003 Design Guideline in chapter Controlling the design • explanation of parameter [pOpenInNewTab] of method neon.openContext • chapter Troubleshooting: New sub-chapter Low performance, referencing AID066 Performance Optimization. • chapter Connecting EntityFields with database columns (RecordContainer): RecordFieldMapping: Renaming of uninitializing option: "Delete" → "Restore Default Value" • minor improvements and bugfixes • This is the official version of the Customizing Manual for ADITO version 2021.2.

Version	Changes
2021.1.1	• new chapter AggregateFields • new chapter Using database views • chapter Deploy extended by paragraph [Deploy_of_a_single_model] • chapter Internationalization enhanced by including a reference to the DeepL API and to the corresponding chapter of the Designer Manual • This is the official version of the Customizing Manual for ADITO versions 2021.1.0 and 2021.1.1.

Version	Changes
2021.0.3	• chapter FilterExtension restructured and extended by subchapters describing properties ○ useConsumer/consumer ○ groupQueryProcess • chapter FilterExtensionSet extended by ○ tip box describing how to relate to a Consumer ○ info box mentioning property groupQueryProcess • chapter Filter presets extended and improved • chapters SingleDataChart and MultiDataChart: Description of new property "colorField" • chapter ViewTemplates: Description of properties common to multiple or all ViewTemplates, including "maxDbRow", "title", and "entityField" • chapter Liquibase update: New note box on emptying the server’s cache after changes in the database structure • chapter GroupLayout: Tip box on how to customize the list items of the View selection button • chapters Add Dashlets and MasterDetailLayout: New info box stating that DashletConfigs are not available for Views having a MasterDetailLayout • appendix "Create Liquibase files automatically" removed and transferred to the Designer Manual (chapter "Plugin Liquibase") • Refactoring because of updated wording: ○ "Diff with DB tables" → "Diff Alias <> DB Table" (see chapter Updating the Alias Definition) ○ "nodes in the dbRecordContainer" → "RecordFieldMappings" (see chapter Connecting EntityFields with database columns (RecordContainer)) • chapter Connecting EntityFields with database columns (RecordContainer): Description of requirement to initialize a RecordFieldMapping before being able to configure it • multiple notes of requirement to initialize an exposed Parameter (under a Provider) before being able to configure it

Version	Changes
2021.0.1	• new chapters on ○ new ViewTemplates Map and MultiEditTable ○ new model type Renderers • new appendix EntityField/Keywords vs. Attributes, explaining the pros and cons of each approach • new appendix $sys variables with overview of all $sys variables and their purposes • new appendix $local variables with description of selected $local variables • description of $local variables extended in chapter JDito system modules and variables • chapter FilterExtensionSet extended by names of $local variables • refactoring and extension of chapter Using keywords (predefined values), because of new keyword data structure (category instead of container) • chapter FilterExtension: bugfix in example code for filterConditionProcess • chapter on ViewTemplate WebContent (IFrame) extended • reference to AID114 "Blueprints" included • description of property [hideContentSearch] included in chapter on ViewTemplate Table • description of new configuration method "user" included in appendix on LoadEntity and WriteEntity • minor improvements and bugfixes • version history table: hyperlinks to chapters added; formatting optimized • This is the official version of the Customizing Manual for ADITO version 2021.0.1.

Version	Changes
2020.2.2	• chapter FilterExtension: new paragraph about examples in ADITO xRM; extended description of variable `$local.comparison` • explanation of property [isLookupFilter] • new tip box regarding [zooming] the code in the Editor window • new example of [CarDriver_entity_contentTitleProcess] using "LoadEntity" • extended information in chapter Troubleshooting • new chapter [EnablingDemoData] covering control and danger of demo data in Liquibase master changelog file • extended description of [contentTitleProcess] and LookupView • chapter System variables: paragraphs on $image and $comp removed • bugfix in initFilterProcess • explanation of the ViewTemplate Generic's property [hideEmptyFields] • tip box added explaining how to [deactivate] a displayValue at runtime • added info box regarding PreviewView in DatalessRecordContainer • minor improvements. • This is the official version of the Customizing Manual for ADITO version 2020.2.2.
2020.2.1	• new chapter DatalessRecordContainer • new chapter Blueprints • chapter Icons: new paragraph about how to find a suitable icon quickly • appendix LoadEntity and WriteEntity: new paragraph "Usage in server processes" • further example of LoadEntity (loading 1 single row) • minor improvements • This is the official version of the Customizing Manual for the ADITO version 2020.2.1.

Version	Changes
2020.2.0.1	• new appendix LoadEntity and WriteEntity, along with warnings regarding permissions to be ignored when using [SqlBuilder] and db.xxx methods • reference to Reporting Manual in description of ViewTemplate Report • note regarding setting of EntityField [ISESSENTIAL] (KeywordEntry_entity) • minor improvements • This is the official version of the Customizing Manual for ADITO version 2020.2.0.
2020.2.0	• new chapters with description of ViewTemplates DynamicSingleDataChart and DynamicMultiDataChart • extended description and illustration of ADITO models and their Logical hierarchy • extended description of Deploy • added example of calculating Person_entity.db.age.value.expression in the expression process of the RecordContainer • example code of FilterExtension improved • minor improvements and bugfixes • This is a beta version of the Customizing Manual for ADITO version 2020.2.0.
2020.1.3.1	• Hotfix: refactoring CHAR(36) → VARCHAR(36) for AB_KEYWORD_ENTRY.KEYID and corresponding EntityFields, including removal of calling autopad functionality • minor improvements • This is the official version of the Customizing Manual for ADITO version 2020.1.3.
2020.1.3	• Syntax of version number changed, in order to have a unique reference to the corresponding ADITO version • This is a beta version of the Customizing Manual for ADITO version 2020.1.3.

Version	Changes
2.3	• Added example code for form definition of ViewTemplate DynamicForm • added short description of ViewTemplate Tiles • refactoring KeywordRegistry_basic → KeywordRegistry_carPool • minor improvements.
2.2	• short notice regarding logging and debugging • warning regarding usage of [openContext] in RecordContainer processes • added chapter "Tree and TreeTable: Advanced explanations" • significant extensions of chapters JDitoRecordContainer, SingleDataChart, and MultiDataChart (see sub-chapters "Advanced explanations") • refactoring SpellingGuidelines → AID001 • refactoring vars.getString(…) → vars.get(…), including revision of chapter System variables, as vars.getString(…) is no longer required • SALUTATION of driver added • field name changed: DRIVINGLICENSEID → DRIVINGLICENSENUMBER • chapter Example: Sum of fines: Note added regarding calculation of fines sum via SQL • minor improvements • version for 2020.1.1
2.1	• New chapters about FilterExtension and FilterExtensionSet; • new chapter on Filter presets • new chapters about ViewTemplates CardTable, DynamicForm, Lookup, and Report • warning regarding database [indices]; specifications for customized icons; examples of database indices in Liquibase file create_carreservation.xml.xml • minor improvements • version for 2020.1.0

Version	Changes
2.0	• new appendix Accessing the value of an EntityField • new appendix Operating state vs. record state • new chapter Resetting Dashboards • chapter IndexRecordContainer extended by instructions on how to rebuild the index • extended warnings of using the [valueProcess] or [displayValueProcess] instead of the "expression" properties • minor improvements • version for 2020.0.2
1.9	• Extended description of ViewTemplates SingleDataChart and MultiDataChart; • extended Liquibase documentation, including how to auto-generate Liquibase files from existing database structure and content (see new appendix "Create Liquibase files automatically) • database changes via Alias Definition • extended description of property "color" • appendix Checklist for new fields added • chapter Actions extended • minor improvements • new version for 2020.0.1
1.8	• Refactoring SqlCondition → SqlBuilder
1.7	• new appendix Requirements for customized Theme • minor further optimizations
1.6	• new chapter 360Degree Context • warning of customizing [xRM_libraries] • new chapter Internationalization • new chapter QuickEntry

Version	Changes
1.5	• refactoring of wording: xRM-Basic → xRM • bugfixes • minor improvements
1.4	• table and illustration fpr dependencies • ER diagram of the ADITO xRM project's core tables (along with their former names, until ADITO 5) of carpool project • new chapter [Colors] • new tip box explaining [mass_edit] support • various minor improvements and bugfixes
1.3	• new appendix Order of execution of Entity processes • bugfixes, minor improvements
1.2	• new appendix Database Access
1.1	• new chapter Layouts • new chapter JDitoRecordContainer • new chapter IndexRecordContainer
1.0	• First release for ADITO version 2019.2.0