Alfabet Data Integration Framework

Alfabet Reference Manual

Ƨ software ^AG

CONVENTIONS USED IN THE DOCUMENTATION

Convention	Meaning
Bold	Used for all elements displayed in the Alfabet interface including, for example, menu items, tabs, buttons, dialog boxes, page view names, and commands. Example: Click Finish when setup is completed.
Italics	Used for emphasis, titles of chapters and manuals. this Example: see the *Administration* reference manual.
Initial Capitals	Used for attribute or property values. Example: The object state Active describes...
All Capitals	Keyboard keys Example: CTRL+SHIFT
File > Open	Used for menu actions that are to be performed by the user. Example: To exit an application, select File > Exit
< >	Variable user input Example: Create a new user and enter <User Name>. (Replace < > with variable data.)
	This is a note providing additional information.
	This is a note providing procedural information.
	This is a note providing an example.
	This is a note providing warning information.

TABLE OF CONTENTS

Chapter 1:   Alfabet Data Integration Framework

The first issue that a company faces when implementing a central IT management software like Alfabet is the integration of existing data about the company IT architecture and management in the database of the IT management software. Usually data has been maintained with different methods in a variety of formats ranging from simple text files or Microsoft® Excel® sheets to databases that store data in different formats. Collecting this data from the different sources and converting them to a format that fits the structure of the inventory for central IT management can be a tedious and time-consuming process.

The second issue that a company might face is that of interfacing with other tools during runtime of the system. Data that is stored in the central inventory might, for example, contain data relevant for accounting that must simultaneously be maintained in a database used by the accounting software. Decisions must be made about where the data should be maintained and how changes are taken over by the other database.

Usually solutions for these issues require the implementation of a variety of technologies that have been customized to interface with each external source. Such solutions typically require significant development effort that must be bought in combination with the standard software and a time- consuming and error-prone conversion process in order to standardize the format of data from diverse sources to match a given import scheme.

Software AG now offers a solution that matches the demand for high flexibility and enables customers to develop their own solutions for data integration without the necessity to convert data to a fixed import format.

The Alfabet Data Integration Framework (referred to as ADIF) is a technology to batch import, export, and manipulate very large amounts of data in the Alfabet database. Its interface is highly configurable by means of native SQL commands in combination with conversion procedures delivered by Software AG. Via ADIF, data can be imported from and exported to the following sources:

* external database tables

* CSV files

* XLS files

* XLSX files

* XML files

The ADIF interface allows data to be imported from any customer-defined format within the data source. Data can be integrated from multiple sources in one step, which means that you can, for example, configure the interface to import data that is partially derived from a database table and partially stored in XML files.

This reference manual describes the process of data integration to the Alfabet database and provides the knowledge required to successfully configure the interface to convert the data provided in customer-specific format to the format required to store the data in the Alfabet database tables. It also describes how to convert Alfabet data for export into a format for further manual or third-party tool-triggered processing outside the Alfabet components.

> ⚠ Please note that ADIF shall only be used to alter data stored in existing database table columns of the Alfabet database. Although SQL commands for creation or deletion of database tables and database table columns are allowed in ADIF to create temporary database tables that only persist

during the import process, you MUST NOT alter the existing database structure. Software AG cannot be held responsible for any damage done to the Alfabet database by customer defined ADIF import or export actions not obeying to this rule.

Advantages of the Alfabet Data Integration Framework

The ADIF capability provides the following advantages:

- Independent of the data format of external data

- Processing of different data formats in one step

- Visual interface that supports configuration of the business logic for import, export, or data manipulation:

  - automatic structure recognition of data to be imported

  - predefined elements guide the developer through the configuration of the required import, export or data manipulation actions

- Functionalities to ease development tasks:

  - editors with support for correct configuration

  - visual debugger that allows the outcome of each step of the process to be controlled during interface configuration including the inspection of intermediary results

  - customizable logging of import, export, or data manipulation steps

  - possibility to create test runs so that the results are only temporarily applied to the database.

Preconditions

These sections describe the preconditions that must be met in order to use ADIF as well as additional sources of information that are available.

Required Skills

The ADIF interface is highly customizable and can be configured by the customer without pre-configuration or additional development from Software AG. The following skills are required to configure the import interface.

- You must know how to write commands in native SQL in order to configure the Alfabet Data Integration Framework. Commands must be written in the SQL version that can be interpreted by the database server hosting the Alfabet database.

- Knowledge about the Alfabet meta-model is required to implement mapping of data to the database tables in Alfabet. See The Alfabet Meta-Model in the Alfabet Database for more information.

Licenses

The Alfabet Data Integration Framework requires a special license agreement with Software AG. ADIF is not included in the base standard functionalities of Alfabet.

Related Documents

It is recommended that you consult the following documents during configuration of the ADIF interface:

- *Documentation of the Alfabet Meta-Model*

A detailed description of all relevant object classes of the Alfabet meta-model and their properties, their meaning within the class model, their relation with other object classes and information about how proper- ties are represented in the database table of the object class. This document helps you to understand the target structure for data import and the source structure for data export or manipulation.

- Reference manual *Configuring Alfabet with Alfabet Expand*

Customization of the standard Alfabet solution may be required when importing data into the Alfabet data- base to provide the base structure required to import the data. For example, custom properties can be added to object classes to store customer-specific data that is not part of the standard meta-model or ac- cess rights can be defined for object classes.

- Reference manual *Configuring Evaluation and Reference Data in Alfabet*

This document provides information about configuration steps that are required before importing data re- lated to cost planning, implementation of roles, and evaluations of objects.

Support

If you encounter any problems during the use of ADIF, please consult Software AG Support:

Please open a ticket in the Empower eService for any service request as well as all non-standard support incidents such as training requests, scripting, or data integration:

https://empower.softwareag.com

When you submit a ticket for a service request, you should include the main release number and patch ver- sion of your Alfabet product. This information can be accessed by clicking **Help** < **About Alfabet**. Tickets will be recorded and transferred to the relevant team.

Empower eService also includes:

- tracking ticket statuses

- local telephone numbers for support.

In addition to the local support telephone numbers, you can use the following toll-free number:

+800 2747 4357

Chapter 2:   The Alfabet Meta-Model in the Alfabet Database

One of the primary tasks in configuring an ADIF scheme is the mapping of external data to the data in the Alfabet database. Data storage in the Alfabet database is determined by the Alfabet meta-model, which is the foundation of the Alfabet solution. The meta-model specifies the object classes used in the Alfabet solution, the properties of the object classes, and the relationships between object classes.

The information in this section provides basic information about the structure of the meta-model and its storage in database tables. It is highly recommended that you read this section in order to understand the detailed description provided about each of the meta-model class in the technical documentation of the meta-model.

The following information is available:

- Understanding the Standard Meta-Model

    - About Object Classes

        - Integrity References

    - About Object Class Properties

- About the Storage of Object Data in the Alfabet Database

    - Database Table Structure and Unique Identifiers

    - Storage of Translatable Object Class Properties

    - Data Types and Formats

    - Storage of Relations Between Object Classes

    - Audit History Storage

- Preconfigured and Customized Restrictions for Data Input

    - Object Class Configuration

        - Object Class Stereotypes

        - Class Keys

        - Mandate Data Handling

    - Object Class Property Configuration

        - Default Values for Object Class Properties

        - Size Restrictions for Object Class Properties

        - Validators for Object Class Properties

        - Protected and Customized Enumerations Assigned to Object Class Properties

    - XML-Based Definitions

    - Reference and Evaluation Data and Class Configuration

Understanding the Standard Meta-Model

The standard meta-model provided by Software AG includes a class model that is made up of object classes and their object class properties. The following information describes information relevant for standard object classes and object class properties. Information that is important to the ADIF process for configured object classes and object class properties is described later in this chapter in the section Preconfigured and Customized Restrictions for Data Input.

The following information is available about the class model:

- About Object Classes

    - Integrity References

- About Object Class Properties

About Object Classes

The Alfabet class model is based on object classes, some of which are visible in the configuration tool Alfabet Expand. Each object class has a set of preconfigured object class properties and attributes.

The class model can be grouped into three different categories of object classes with regard to their relevance for data import:

- The extendable meta-model that represents instance object classes.

These classes are used to persistently store data about the customer's IT landscape in the Alfabet database. They are displayed in the explorer of the configuration tool Alfabet Expand. Most of these classes are customizable and customers can add custom properties to the classes to store company specific data that is not reflected in the standard properties of the object class. The extendable meta-model is the main target for data import via the ADIF scheme.

- Object classes that represent objects designed to support functionality.

These classes are used to persistently store data about object classes supporting functionality. This includes, for example, the class `TimeStatus` that is used to store the lifecycle of objects or the class `RoleTypeConfig` that is used to store information about which roles a user can have with regard to an object. These object classes are also relevant for data import via the ADIF scheme. Only the most important object classes for storage of functionality-related data are visible in the explorer in Alfabet Expand, but the documentation of the meta-model provides detailed information about all these classes.

- Auxiliary constructs required for Alfabet functionality.

These object classes are not relevant for data import and therefore not described in the meta-model documentation. Some of the auxiliary object classes are persistently stored in database tables, whereas others are not persistently stored.

> ⚠ Please note that object classes having the attribute **Automatically Managed** set to `true` are object classes that shall only be changed by mechanisms triggered by the Alfabet software components and must not be changed by ADIF import or any third party component.

In the **Meta-Model** tab in the configuration tool Alfabet Expand, the **Class Model** explorer displays a set of object classes that are relevant to configuration of the Alfabet solution. The object classes and their object class properties are visualized as nodes in the explorer structure.

You may see any of the following under the **Classes** node:

- A private class has a red lock icon and cannot be modified. Custom properties cannot be configured for private classes.

- A protected class has an orange lock icon and can be edited in a limited way. A protected class typically has private properties that cannot be edited but allows for custom properties to be created and configured for the class.

- A public class has no lock displayed on the icon. This is a custom class that has typically been created by your enterprise in the configuration tool Alfabet Expand. All relevant attributes may be edited, and the custom class may be deleted. For custom classes created and defined by your enterprise, the attribute **Tech Name** must be explicitly defined.

For each object class displayed in the Alfabet Expand interface (as well as those that are not visible), a database table exists. Each class property displayed below its object class node corresponds to a column in the database table.

If an object class node or an object class property node is selected in the explorer structure, the attributes of the class/property will be displayed in the right pane in the Alfabet Expand interface. These attributes specify the technical data about the object class/object class property that is used to build the database table and to process the object data within the Alfabet solution.
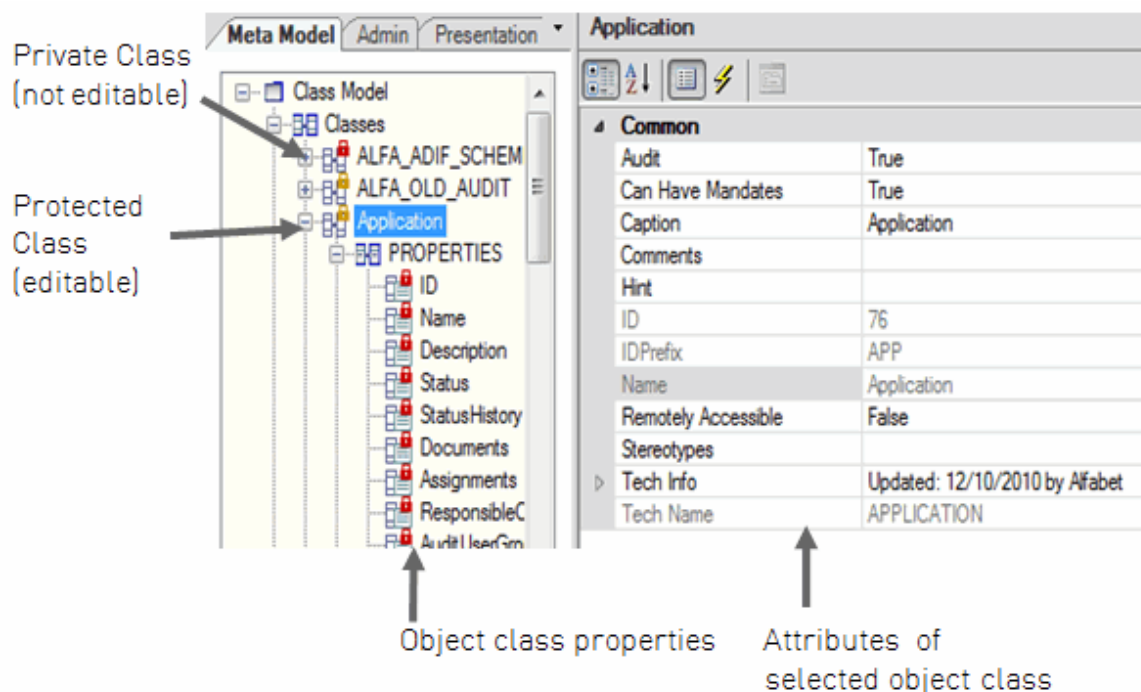
FIGURE: Attributes and properties of the object class Application displayed in the explorer of the tool Alfabet Expand

The object class within the Alfabet solution are identifiable by the following three attributes:

- **Name**: The attribute **Name** is unique for each object class in the meta-model as well as for each property in the object class. The attribute **Name** is used to display the object classes and object class properties in the Alfabet Expand user interface and to specify object classes and object class

properties in the Alfabet query language. (The Alfabet query language is used to search and retrieve data from the database.)

- **Tech Name**: The attribute **Tech Name** is unique for each object class and specifies the database table name for the object class. The attribute **Tech Name** of an object class property specifies the name of the database table column for the property. The **Tech Name** is therefore relevant to identify object classes and their properties in native SQL queries when importing data via the ADIF scheme.

- **Caption**: The attribute **Caption** specifies the name of the object class or object class property displayed in the Alfabet interface.

The documentation of the standard meta-model includes the relevant information for each object class. For some object classes, the data that may be input to the database table may be restricted due to customizations made to the object class. Such customizations include, for example, the configuration of object class stereotypes, class keys requiring unique data input, and the implementation of the mandate concept.

When importing data via the ADIF scheme, such restrictions must be taken into consideration when mapping and importing data to the database, otherwise the data import may not be valid. For an overview of potential configurations that may impact the mapping of object classes in the ADIF scheme, see the section Preconfigured and Customized Restrictions for Data Input.

Integrity References

In the Alfabet meta-model, objects in an object class may have a dependent relationship on other objects in other classes. This is described in the **Integrity Info** attribute.

If an object is deleted, all dependent objects of dependent object classes will also be deleted. For example, if an application is deleted, all information flows for which this application is specified in the Owner, To or From property of an object are also deleted.

The documentation of the standard meta-model includes the relevant dependency information for each object class.

> ⚠ Please note that a default value is set for the **Apply Base Integrity** attribute for private and protected object classes. The value `True` is set if base integrity management is performed for the private or protected object class. In this case, if an object of the object class is deleted all references to the deleted object will also be deleted. If the value `False` is set, no integrity management will be performed for the object class.
>
> In order to improve performance, the value `False` may be set for some custom object classes if no object classes reference the custom object class. However, it is highly recommended that you contact Software AG Support if you want to set the value to `False` for a custom object class. Setting the **Apply Base Integrity** property to `False` may result in serious inconsistency problems in the database.

About Object Class Properties

Most object classes in the meta-model have object class properties. You may see any of the following under the **Properties** node:

- A private property has a red lock icon and cannot be modified.

- A protected property has an orange lock icon. This is a standard property that allows some property values to be edited including, for example, the **Caption** attribute and the **Hint** attribute. A standard property may not be deleted.

- A public property has no lock displayed on the icon. This is a custom property that has typically been created by your enterprise in the configuration tool Alfabet Expand. All relevant attributes may be edited, and the custom property may be deleted. For custom properties created and defined by your enterprise, the attribute **Tech Name** has to be explicitly defined.

Each object class property has an attribute **Property Type** that specifies the data type of the property (for example, String, Date, Integer, Reference, etc.). The property type definition will determine what other attributes can be defined for the object class property. For example, for a property of the type String, you can optionally define the **Default**, **Enum**, **Size**, or **Validator** attributes. For an overview of the data types possible for the **Property Type** attribute, see the section Data Types and Formats.

FIGURE: Attributes of the object class property StartDate of the object class Project displayed in the explorer of the tool Alfabet Expand

The documentation of the standard meta-model includes the relevant information for each object class property. For some property types, the data that may be written to the database table may be restricted due to additional meta-model or solution configurations. Such customizations include, for example, configuration of data input validators, enumerations, reference data, or specifications made in XML objects.

When importing data via the ADIF scheme, such restrictions must be taken into consideration when mapping and importing data to the database, otherwise the data import may not be valid. For an overview of

potential configurations that may impact the mapping of object classes in the ADIF scheme, see the section [Preconfigured and Customized Restrictions for Data Input](#).

> (i) For more information about defining custom properties, see the section *Configuring Custom Properties for Protected or Public Object Classes* in the reference manual *Configuring Alfabet with Alfabet Expand*.

About the Storage of Object Data in the Alfabet Database

A database table exists in the Alfabet database for each persistent object class. Additional database tables store relationships between objects as well as the audit history of objects.

The following information is available:

*   [Database Table Structure and Unique Identifiers](#)

*   [Storage of Translatable Object Class Properties](#)

*   [Data Types and Formats](#)

*   [Storage of Relations Between Object Classes](#)

*   [Audit History Storage](#)

Database Table Structure and Unique Identifiers

The name of the database table is specified by the **Tech Name** attribute of the object class. The names of the columns in the table are specified by the **Tech Name** attributes of the object class properties of the respective object class. For most standard Alfabet object classes and properties, the **Tech Name** is identical to the **Name** attribute of the object class or property. All **Tech Name** attributes are stored in upper case letters in Alfabet (<TECHNAME>).

If the technical name derived from the object class name conflicts with keywords reserved for the relational database management system (RDBMS) of the Oracle or Microsoft SQL Server, the **Tech Name** will be written with a prefix "T_" for tables of object classes and a prefix "A_" for columns representing attributes.

In addition to the columns for object class properties, a database table has several additional columns that are automatically added by the Alfabet software when an object is added to the database. Values for these columns may NOT be included in the data import for new objects. However, such columns can be used to identify existing objects:

*   **REFSTR**: The REFSTR is a unique reference string that identifies the object in the database. The REFSTR is used to store relations between objects. During data import via the ADIF scheme, REFSTRs for new objects must be set before a relation between a new object and any other object is created. The mechanism for setting REFSTRs is part of the ADIF import scheme mechanics.

*   **INSTGUID**: The INSTGUID is a unique attribute that can be used to identify the object across databases. The INSTGUID is even unique with regard to different databases. If objects are added to two different databases, they may have the same REFSTR and ID in both databases, but the INSTGUID is different. During data import via the ADIF scheme, INSTGUIDs for new objects are set automatically and must not be included in the import specification.

- **ALFA_INSTID**: The `ALFA_INSTID` is a subset of the `REFSTR` property. During data import via the ADIF scheme, `ALFA_INSTID`s for new objects are set automatically. The `ALFA_INSTID` may not be included in the import specification.

- **ID**: An `ID` is specified for some object classes that are visible in the Alfabet interface in order to help users working in the Alfabet interface to identify or search for objects. For visible object classes for which the property `ID` exists, the `ID` is filled automatically. The `ID` attribute consists of a prefix that is based on the value of the attribute **ID Prefix** of the object class and a number. During data import via the ADIF scheme, `ID`s for new objects are set automatically and should NOT be included in the import specification.

	REFSTR	INSTGUID	ID	NAME	NAME_1031	
	76-2518-0	117F79F6CC694...	APP-2518	Business EAI Pla...	*NULL*	*N*
	76-2525-0	C88B1BFE80A24...	APP-2525	Groupware Servi...	*NULL*	*N*
	76-2538-0	9447219E743F4...	APP-2538	Mafo-Portal	*NULL*	*N*
	76-2566-0	1B02FDB074924...	APP-2566	PS Global	*NULL*	*N*
	76-2576-0	49F859F0A5414...	APP-2576	ALLFiance PISA	*NULL*	*N*

Objekt-Explorer: dbo.ALFADBSYS, dbo.ALFAINSTID, dbo.APPGROUPMANA, dbo.APPLICATION, dbo.APPLICATION_AU, dbo.APPLICATIONDIA

FIGURE: Database table for the object class Application showing attributes that are automatically added to the table

Storage of Translatable Object Class Properties

One language is always configured as the default language in Alfabet. However, if your enterprise requires additional language options for the Alfabet interface, your solution designer will configure relevant cultures for the enterprise in the configuration tool Alfabet Expand.

If the attribute **Support Data Translation** is set to `True` for a culture, then the database tables will include columns for translation purposes for some object classes. For object classes that are visible in the Alfabet interface, some object class properties are translatable, and users can specify translations for properties (mainly the `Name` and `Description` properties) in the relevant object class editor.

The translations are stored in separate columns in the database table with the column name <property Tech Name>_<language code>. For example, the `Name` property of the class Application can be translated to German. The number 1031 is the language code for German. Therefore, the column **NAME_1031** is automatically added to the database table. Please note that language columns are only available in database tables if the attribute **Support Data Translation** is set to `True` for a culture.

Data Types and Formats

Each object class property has an attribute **Property Type** that specifies how data must be defined (for example, as a string, date, integer, reference, etc.). The **Property Type** determines how the data is stored in the database tables. The data storage mode may be different for Oracle® database servers and Microsoft® SQL Servers®.

Some property types may be further configurable via other attributes of the object class property. For example, for a property of the **Property Type** `String`, you can optionally define an enumeration that specifies values that are allowed to be used for the string property. Likewise, many property types allow a default value. The value specified for the default value is automatically written to the database if the user does not explicitly define a value for the property in the Alfabet interface. For more information about the

possible dependencies for object class properties, see the section [Preconfigured and Customized Re-strictions for Data Input](#).

The following table provides an overview of the permissible property types for object class properties, the data storage mode for both Microsoft® SQL Server® and Oracle® database servers, and additional information relevant for the property type.

Property Type	Stored in MS SQL Server® Databases As:	Stored in Oracle Databases As:	Additional Information
Boolean	bit	NUMBER	A default value can be defined via the **Default** attribute for properties of the **Property Type** Boolean. Allowed values: 1|0 (1 = true, 0 = false)
Date	datetime	TIMESTAMP	
DateTime	datetime	TIMESTAMP	
Integer	int	NUMBER	An enumeration can be defined via the **Enum** attribute for properties of the **Property Type** Integer. A default value can be defined via the **Default** attribute for properties of the **Property Type** Integer. For more information, see [Preconfigured and Customized Restrictions for Data Input](#).
Real	float	FLOAT	An enumeration can be defined via the **Enum** attribute for properties of the **Property Type** Real. A default value can be defined via the **Default** attribute for properties of the **Property Type** Real. The number of decimal places allowed can be defined via the **Precision** attribute for properties of the **Property Type** Real. For more information, see [Preconfigured and Customized Restrictions for Data Input](#).
RealArray	nvar-char(max)	NCLOB	An enumeration can be defined via the **Enum** attribute for properties of the **Property Type** RealArray.
Reference	varchar(20)	VARCHAR2(20)	The **Type Info** attribute must be defined for a property of the **Property Type** Reference.

Property Type	Stored in MS SQL Server® Databases As:	Stored in Oracle Databases As:	Additional Information
ReferenceArray	nvarchar(max)	NCLOB	The data is stored as `NCLOB` in the database table of the object only if the **Reference Support** attribute = "false". If the **Reference Support** attribute = "true", the data is stored in the `RELATIONS` table. The **Type Info** attribute must be defined for a property of the type `ReferenceArray`. For more information about the impact of the **Reference Support** attribute on data storage as well as additional details about the storage of relations, see Storage of Relations Between Object Classes
String	varchar($Size$)	NVARCHAR($Size$)	An enumeration can be defined via the **Enum** attribute for properties of the type `String`. A required format for input values can be defined for properties of the type `String`. A default value can be defined via the **Default** attribute for properties of the type `String`. The maximum amount of permissible characters may be defined via the **Size** attribute for properties of the type `String`. For more information, see Preconfigured and Customized Restrictions for Data Input.
StringArray	nvarchar(max)	NCLOB	An enumeration can be defined via the **Enum** attribute for properties of the type `StringArrays`. For more information, see Preconfigured and Customized Restrictions for Data Input.
Text	nvarchar(max)	NCLOB	
Time	datetime	TIMESTAMP	
Url	nvarchar(max)	NCLOB	The data is stored as a title text and the URL divided by a line break written as character code 13 followed by character code 10. If no title is provided, the definition must start with the linebreak. For example: `Title`

Property Type	Stored in MS SQL Server® Databases As:	Stored in Oracle Databases As:	Additional Information

			`http://target.com`
			
			which is identical to
			
			`TitleCHAR(13)CHAR(10)http://target.com`

Storage of Relations Between Object Classes

The attribute `REFSTR` is a unique internal object ID that explicitly identifies each object in the database and is used for references between objects in the database.

In database columns of object class properties of the data type `Reference`, the `REFSTR` of the referenced object is stored as the data type varchar(20) in an SQL database and as the data type `VARCHAR2(20)` in an Oracle database.

Object class properties of the type `ReferenceArray` are handled in two different ways depending on the setting of the attribute **Reference Support** of the object class property:

- If **Reference Support** is set to `true`, the relations specified by the property are stored in the `RELATIONS` table. No database column is available for the property in the table of the object class itself. The `RELATIONS` table specifies the relation in the following database columns:

    - **FROMREF**: The `REFSTR` value of the object that is referencing another object. It is stored as the data type `varchar(20)` in an SQL database and as the data type `NVARCHAR2(20)` in an Oracle database.

    - **PROPERTY**: The property of the object specified with **FROMREF** that defines the relation between the objects. It is stored as the data type `nvarchar(64)` in an SQL database and as the data type `VARCHAR(64)` in an Oracle database.

    - **TOREF**: The `REFSTR` value of the referenced object It is stored as the data type `varchar(20)` in an SQL database and as the data type `NVARCHAR2(20)` in an Oracle database.

If references stored in the RELATIONS table are to be imported via an ADIF scheme, the RELATIONS table is filled automatically by the ADIF mechanism if a specified data format is fulfilled.

- If **Reference Support** is set to `false`, the relations are stored directly in the database table of the object as a string stored as the data type `nvarchar(max)` in an SQL database and as the data type `NCLOB` in an Oracle database. The `REFSTR` of all objects referenced by the property are listed whitespace separated.

Object class properties of the data type `Reference` or `ReferenceArray` can target only objects in a defined subset of object classes in the Alfabet meta-model. This subset is defined by the attribute **Type Info** of the object class property defining the reference or reference array. The attribute **Type Info** of the object class property specifies the name of the allowed target object classes as a comma-separated list.

Audit History Storage

By default, the audit history is displayed per object class on the Alfabet interface. The user can access the change history of a selected object by clicking the **Audit Trail** 👣 button. You can create native SQL-based configured reports to extract other information from the audit history such as user-centric change tracking, history trend reporting, reporting on new and changed objects (for example, applications), information quality management.

Audit information is stored in a separate database table for each object class. For each object class for which the attribute **Audit** is set to "`true`", a table named <ObjectClassTechName>_AU is created. Whenever an object is changed, a new line is written to the audit table that stores information about the change that has been performed, the user who has performed the change, and the time that the change was made.

> ℹ️ For information about the structure of the database table for auditing, see the section *Defining Audit Management Related Configured Reports* in the reference manual *Configuring Alfabet with Alfabet Expand*.

During import via ADIF, the audit history tables of the object classes for objects that have been changed, created or deleted via the ADIF procedure are automatically filled with the appropriate information. The user who is triggering the ADIF import is defined as the user performing the change.

Some of the mechanisms required for the ADIF import lead to changes to database columns that have no column in the audit table (for example, for **INSTGUID**). When a change is performed to the object, a new line is added to the audit table. This line indicates that a change has been made, but the type of change is not visible in the audit table because all object class properties remain unchanged. The ADIF interface allows the import scheme to be configured so that the lines with no visible change are automatically delete from the audit table.

> ⚠️ The ADIF import definitions should never include the audit tables. Changing the audit table for an object can lead to severe database inconsistencies and may result in an error occurring.

Preconfigured and Customized Restrictions for Data Input

Alfabet allows for a significant amount of customization in order to configure the software to the needs of your enterprise. In addition to the customizations of the class model including, for example, the configuration of object class stereotypes or the creation and definition of custom properties, a variety of other configurations may be required to support various functionalities in Alfabet.

- The configuration of reference and evaluation data determine the permissible values for some object class properties. Evaluations and their indicators, cost and income types, role types, and diagram views are examples of some of the objects that are based on configurations that may be assigned to a set of object classes used in your enterprise. Using a value other than the values specified for the object class in the **Configuration** module may result in an error occurring.

- The configuration of XML-based objects in Alfabet Expand determine the permissible data input for some object classes or object class properties. For example, multiple object classes have a property `Status` that allows the release status of the object in the approval process to be defined. The possible values for release status definitions are configured in the XML object **ReleaseStatusDefs**. Using a value other than the values specified for the object class in such an XML definition may result in an error occurring.

⚠️ **The customization of the meta-model must be performed prior to importing data via the ADIF scheme.**

It is imperative to the success of the integration procedure that all customized configurations are communicated to the parties involved in the data mapping process. For each customization made to the meta-model, it is highly recommended that your enterprise validates the correctness of the mapping of impacted objects in the ADIF schema. Otherwise, potential malfunction may result.

For detailed information about the potential configurations that can be made to the Alfabet solution, see the reference manual *Configuring Alfabet with Alfabet Expand* and the reference manual *Configuring Evaluation and Reference Data in Alfabet*.

The following sections discuss potential areas of customization of Alfabet. If your solution configuration includes such customizations, your enterprise's meta-model will be different from the standard meta-model documented here.

Potential areas of customization include:

- Object Class Configuration

  - Object Class Stereotypes

  - Class Keys

  - Mandate Data Handling

- Object Class Property Configuration

  - Default Values for Object Class Properties

  - Size Restrictions for Object Class Properties

  - Validators for Object Class Properties

  - Protected and Customized Enumerations Assigned to Object Class Properties

- XML-Based Definitions

- Reference and Evaluation Data and Class Configuration

Object Class Configuration

For some object classes, the data that may be written to the database table may be restricted due to customizations made to the object class. The documentation of the meta-model includes information about the restrictions that apply to each object class in the standard meta-model.

Such restrictions must be taken into consideration when mapping to the database, otherwise the data import may not be valid. It is your responsibility to be informed about the customizations in your enterprise's solution configuration that are relevant to the integration procedure.

The following configuration issues are relevant to object classes:

- Object Class Stereotypes

- Class Keys

- [Mandate Data Handling](#)

Object Class Stereotypes

An object class stereotype is a sub-classification within an object class. By configuring object class stereo-types, it is possible to specify an object class to have multiple class types, each of which captures a speci-fied set of object class properties, reference data, and class configurations.

Stereotypes can only be configured for the object classes Application, Component, Device, ICT Object, Do-main, and Project. The solution designer creates the object class stereotypes for these object classes in an XML definition via the attribute **Stereotype** on the object class.

These object classes also have an object class property **Stereotypes** of the data type `String`. When add-ing objects to the database via the integration procedure, the value for the object class property **Stereo-type** must be identical to the stereotype name of one of the stereotypes defined in the attribute **Stereo-type** of the object class.

It is your responsibility to be informed about customized object class stereotypes in your enterprise's solu-tion configuration that are relevant to the integration procedure.

Please note the following information that is relevant to the implementation of object class stereotypes in your enterprise:

- For the object classes `Project` and `Domain`, the configured object class stereotypes must be hierarchically structured and specified. These stereotypes require the additional configuration of XML objects. For more information about the additional configuration requirements required for the object classes **Project** and **Domain**, see the sections *Configuring the Project Management Capability* and *Configuring Domain Models and Domain Planning* in the reference manual *Configuring Alfabet with Alfabet Expand*.

- A stereotype framework can also be configured for value nodes implemented in the **Strategy Deduction** functionality. In this case, the configuration of stereotypes is not made on the object class but rather in the XML object **ValueManager**. When adding objects to the database via the integration procedure, you must ensure that the value for the object class property Stereotype is identical to the stereotype name of one of the value node stereotypes defined in the XML object **ValueManager**.

For more information about this configuration, see the section *Configuring the Strategy Deduction Capabil-ity* in the reference manual *Configuring Alfabet with Alfabet Expand*.

Class Keys

Class keys are configurable for artifact object classes. A class key with an attribute **Unique** set to `True` specifies one or a combination of object class properties that must be unique for that class. If the data does not fulfill the requirements specified in the class key definition, the object cannot be created in the data-base.

For example, a class key for the object class `Application` defines that the combination of the Name and Version properties must be unique. When a user tries to create an application via the Alfabet interface that is identical in both name and version with an existing application in the

Alfabet database, the attempt to create the application is rejected and an error message informs the user that the object already exists.

The Alfabet meta-model also allows class keys to be specified that do not require uniqueness. In this case, the class key attribute Unique will be set to `False`. The purpose of such a class key is to speed up the search functionality by creating an index for each class key. If the class key attribute `Unique` will be set to `False`, the class key is NOT relevant for data import via the integration procedure.

It is your responsibility to be informed about configured object class stereotypes relevant to the persons responsible for the integration procedure.

It is your responsibility to be informed about the customized class keys in your enterprise's solution configuration that are relevant to the integration procedure When importing data via ADIF, import of data that leads to a class key violation will cause an error to occur and the import process to stop.

> (i) For more information about class keys, see the section *Configuring Class Keys for Object Classes* in the reference manual *Configuring Alfabet with Alfabet Expand*.

Mandate Data Handling

The mandate capability in Alfabet is a means to organize and structure the federated architecture of holding companies and federated enterprises. The assignment of mandates to users and objects allows the holding company to structure the objects in the enterprise architecture and regulate visibility and accessibility to common objects across some or all business units.

The assignment of objects to mandates is stored in the database column MANDATEMASK in the database tables for all object classes that can have mandates in the Alfabet database. The mandate mask is an integer that represents a bit string calculated based on the number and kind of mandates assigned to an object.

When a company specifies a new mandate, an object of the class ALFA_MANDATE is created. Each mandate has an ID property. The ID is automatically assigned to the new mandate. The ID is a running number starting with 1 that defines the position of the mandate in the bit string of the mandate mask.

The mandate mask consists of one bit for each mandate. When the object is assigned to a mandate, the bit on the position of the mandate in the mask is set to 1. If the object is not assigned to a mandate, the bit on the position of the mandate is set to 0. The first bit in the bit string (the position 0) is set to 1 if the object is not assigned to any mandate. For objects assigned to a mandate, the first bit in the bit string is set to 0.
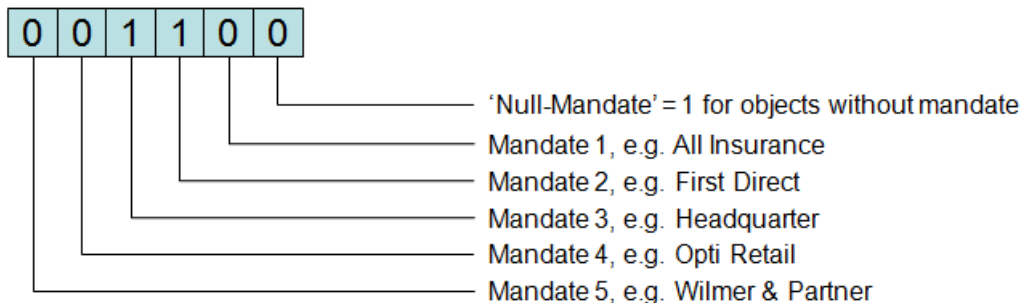
FIGURE: Example of a mandate mask in a configuration with 5 mandates. The object is assigned to mandate 2 and 3.

To evaluate whether an object is visible to a user, the mandate mask of the object is compared bit-wise with a mandate mask generated from the mandate settings of the current user.

When the mandate-related access permissions for an object are evaluated, the mandate mask of the current object is compared with the mandate mask of the current mandate that the user is logged in with. This mandate mask consists of one bit for each mandate. The bit on the position of the current mandate and the first bit in the bit string is set to 1.

⚠️ The mandate mask generated to evaluate the current access permissions of the user is not identical to the mandate mask of the user stored in the property `MANDATEMASK` of the object class `Person`. The mandate mask stored with the property `MANDATEMASK` defines the access permissions for other users to the data about the user stored in the Alfabet database. The current user mandate mask generated to evaluate access permissions is not stored in the database but generated during evaluation of access permissions as a parameter describing the current environment.

During evaluation of access permissions, the current user mandate mask and the object mandate mask are compared bitwise. If the bit on a position is 1 for both object and user, the resulting bit on the position is 1. If either the user or the object or both has a bit value of 0, the bit on the position is set to 0.

The result is a bit string representing an integer. If the bit string contains at least one bit set to 1, the resulting integer is higher than 0 and the object is visible to the user.
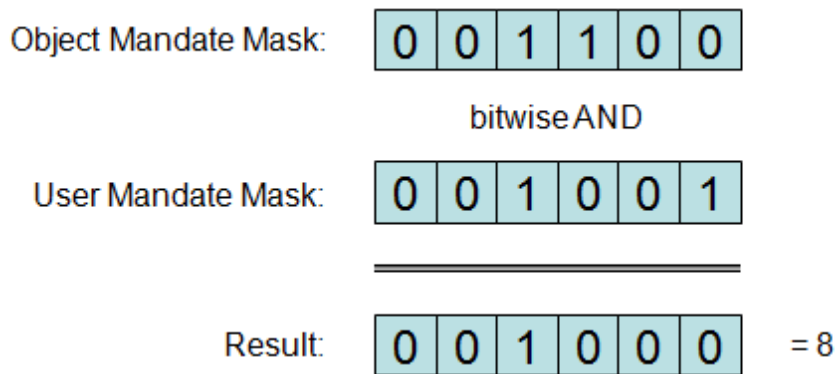
FIGURE: Calculation of the permission to an object based on the mandate masks of the current user mandate and the object

Object Class Property Configuration

For some property types, the data that may be input to the database table may be restricted due to additional meta-model or solution configurations. When importing data via the ADIF scheme, such restrictions must be taken into consideration when mapping and importing data to the database, otherwise the data import may not be valid.

The documentation of the standard meta-model includes restrictions preconfigured by Software AG that apply to each object class property. Furthermore, relevant information is provided if a customized configuration is possible that may impact the data input. In this case, it is your responsibility to be informed about customizations in your enterprise's solution configuration that are relevant to the integration procedure.

The following configuration issues are relevant to object class properties:

*   Default Values for Object Class Properties

*   Size Restrictions for Object Class Properties

- [Validators for Object Class Properties](#)

- [Protected and Customized Enumerations Assigned to Object Class Properties](#)

Default Values for Object Class Properties

Default values can be specified for object class properties of the **Property Type** `String`, `Real`, `Integer`, and `Boolean`.

If a default value is defined for an object class property via the attribute **Default Value**, the default value will be entered in the database table column for the object class property for each new object if no other value has been specified by the user creating the object.

Default values for custom object class properties are not set automatically during import. When importing data to the Alfabet database via the ADIF scheme, the import definition must specify the default value for custom object class properties that do not have a relevant value defined.

It is your responsibility to be informed about default values configured for custom object class properties in your enterprise's solution configuration.

Size Restrictions for Object Class Properties

Size restrictions can be specified for the value defined for object class properties of the **Property Type** `String`. If a value is defined for the attribute **Size** of an object class property, the variable `$Size$` for the `varchar` to store the property equals **Size** +1.

Validators for Object Class Properties

Validators can be specified for object class properties of the **Property Type** `String`. The attribute **Validator** of the object class property allows a regular expression to be specified that enforces a required format for that property.

It is your responsibility to be informed about custom validators configured for object class properties in your enterprise's solution configuration.

> (i) For more information about the syntax conventions for regular expressions, see http://msdn.microsoft.com/en-us/library/hs600312(VS.71).aspx.

Protected and Customized Enumerations Assigned to Object Class Properties

Enumerations allow a range of permissible values for a property to be defined. Enumerations can be defined and specified for object class properties of the **Property Type** `String`, `StringArray`, and `Integer` in Alfabet Expand. An enumeration can be reused for multiple object class properties for multiple object classes.

In Alfabet Expand, existing protected and custom enumerations are visible in the **Meta-Model** tab under the explorer node **Enums**. Each enumeration has a **Name** attribute, and the permissible values are defined

via the **Items** attribute. Once the enumeration is assigned to the relevant object class property via its **Enum** attribute, the permissible values that users can select for an object class property are restricted to the values defined for the enumeration.

Protected enumerations are part of the standard meta-model and have been preconfigured by Software AG. These enumerations cannot be deleted. However, the permissible values that are defined in the **Items** attribute may be edited. If the configuration of a protected enumeration has been customized by your enterprise, then the permissible values that can be mapped to the associated object class property will differ from the standard permissible values described in the meta-model documentation.

It is your responsibility to be informed about customizations made to protected enumerations in your enterprise's solution configuration.

> ⓘ For information about the configuration of protected and custom enumerations, see the section *Defining Protected and Custom Enumerations* in the reference manual *Configuring Alfabet with Alfabet Expand*.

XML-Based Definitions

For some object class properties, the permissible data input is restricted to values that are coupled with an XML object that must be configured in Alfabet Expand. For example, multiple object classes have a property Status that allows the release status of the object in the approval process to be defined. The possible values for release status definitions are configured in the XML object **ReleaseStatusDefs**. Using a value other than the values specified for the object class in such an XML definition may result in an error occurring.

It is therefore highly recommended that you consult the meta-model documentation prior to mapping and importing values for an object class property via the ADIF scheme. The meta-model documentation provides detailed information about potential dependencies between XML object configurations and the definition of input values for object class properties.

It is your responsibility to be informed about customizations made to XML object definitions in your enterprise's solution configuration.

> ⓘ For information about the configuration of XML objects, see the section *Working with XML Objects* in the reference manual *Configuring Alfabet with Alfabet Expand*.

Reference and Evaluation Data and Class Configuration

The permissible values for some object class properties are determined by the configurations made for the associated object class. These configurations are carried out in the **Configuration** module of Alfabet. Evaluations and their indicators, cost and income types, role types, and diagram views are examples of some of the objects that are based on configurations that may be assigned to a set of object classes used in your enterprise.

Such configuration objects are created independent of an object class. The configuration objects must then be assigned to the object class(es) that they are relevant for. This allows you to create reference or evaluation data that is relevant to a specific object class context. For example, indicators required to evaluate applications typically differ from the indicators required to evaluate devices.

It is therefore highly recommended that you consult the meta-model documentation prior to mapping and importing values for an object class property via the ADIF scheme. The meta-model documentation provides detailed information about potential dependencies between configuration objects and the definition of input values for object class properties.

It is the responsibility of your enterprise's solution designer to convey any information about relevant configuration objects to the persons responsible for the integration procedure.

Chapter 3:   Configuring ADIF Schemes

This chapter provides general information about how to work with the interfaces that allow you to configure ADIF schemes. The process for data import and export and the required structure and configuration of ADIF schemes for the import and export of data is different. Therefore, the detailed description of how to define data manipulation via ADIF is described in separate chapters that presume knowledge about the configuration basics described in this chapter:

- *Configuring Data Import with ADIF*

- *Configuring Data Export with ADIF*

The ADIF scheme is XML based. Configuration of the ADIF scheme can be performed in a user interface designed by Software AG or directly in a text editor. Both options are described below.

Configuring ADIF via the ADIF Explorer

ADIF import and export schemes can be created and managed in the tool Alfabet Expand or in a separate ADIF user interface. In both cases the design is identical.

ADIF import and export schemes are managed via an explorer tree. The explorer has two nodes on the root level: **ADIF Schemes** and **Meta-Model**. The **ADIF Schemes** sub-tree allows you to create and edit ADIF import and export schemes. The **Meta-Model** tab provides information about the Alfabet meta-model in order to help you to define your ADIF schemes.

The ADIF Schemes Sub-Tree of the ADIF Explorer

Each ADIF import or export scheme that you create will be listed in the explorer below the root node **ADIF Schemes**. The ADIF scheme is configured by adding command elements to the explorer as sub-elements of the ADIF scheme and by specifying the relevant attributes of these sub-elements.

> ⓘ All changes performed in the ADIF explorer of Alfabet Expand are automatically saved to the Alfabet database. The **Save** button is not required to save the changes to the ADIF scheme.

Adding and Configuring ADIF Scheme Elements

New ADIF scheme elements can be added to the explorer tree using the context menu available for the existing explorer nodes. The method to add a new ADIF scheme element is the same, regardless of the element type (for example, attributes or SQL commands). To add a new ADIF scheme element to an existing explorer node:

1) In the explorer, right-click the explorer node element to which you want to add a new ADIF scheme element and select **Create** < **include option** >. The context menu only offers the options that are applicable for the selected explorer node element.

The explorer displays the name of the explorer node element and an icon that indicates the element type. The icon has a green mark to indicate that the new explorer node element is active. New explorer node

elements are active by default. You can deactivate any explorer node elements to inhibit execution by set-
ting the attribute Is Active of the node to False. The explorer node element is then displayed with a red
mark:

FIGURE: Active and inactive ADIF scheme nodes

When an XML element node is expected to have a high number of element types as child nodes, folder
nodes are automatically added to the XML element to structure the content. These folders are only availa-
ble in the explorer and do not have any impact on the storage of data in the XML.

When you click a node in the explorer, the attributes of the XML element represented by the node are dis-
played in the property window on the right of the explorer tree. You can edit all attributes displayed in
black. Attributes displayed in grey are view only.
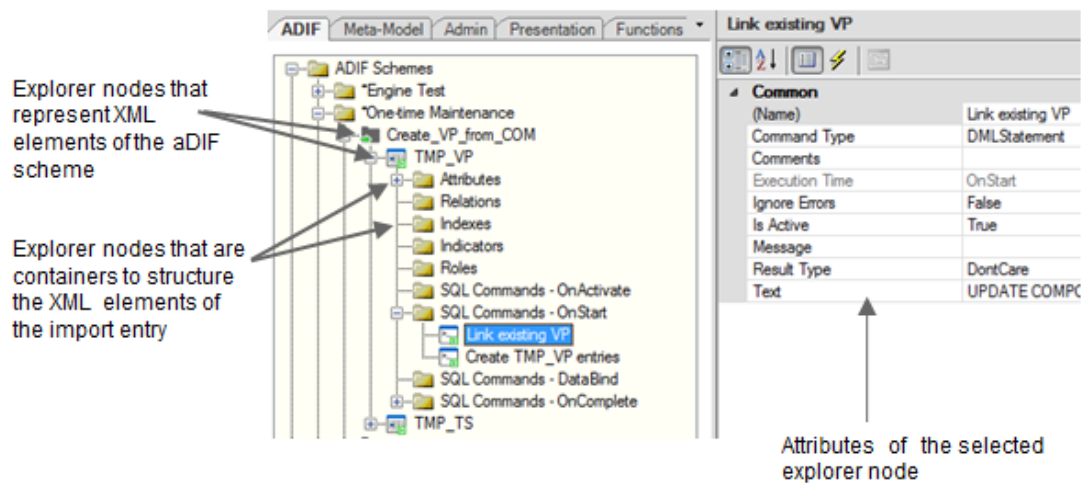
FIGURE: The ADIF Schemes sub-tree of the ADIF explorer

When you create a new ADIF scheme element, the ADIF scheme element is public, that means it is com-
pletely editable. Public Alfabet provides predefined standard ADIF schemes for special import actions, for
example in the context of interfaces with third party applications. These predefined ADIF schemes are ei-
ther private, that means that they cannot be changed at all, or protected, which means that you can edit
only a subset of the attributes of the preconfigured child elements of the ADIF scheme. It is possible to add
new ADIF elements to protected ADIF schemes. The ADIF elements added by the customer are public and
completely editable. Private and protected ADIF scheme have a lock symbol indicating their editability sta-
tus. The lock is orange for protected ADIF schemes and red for private ADIF schemes. Within a protected
ADIF scheme, editability of the attributes is visible in the attribute windows of the child elements only. De-
activated attributes are displayed in grey.
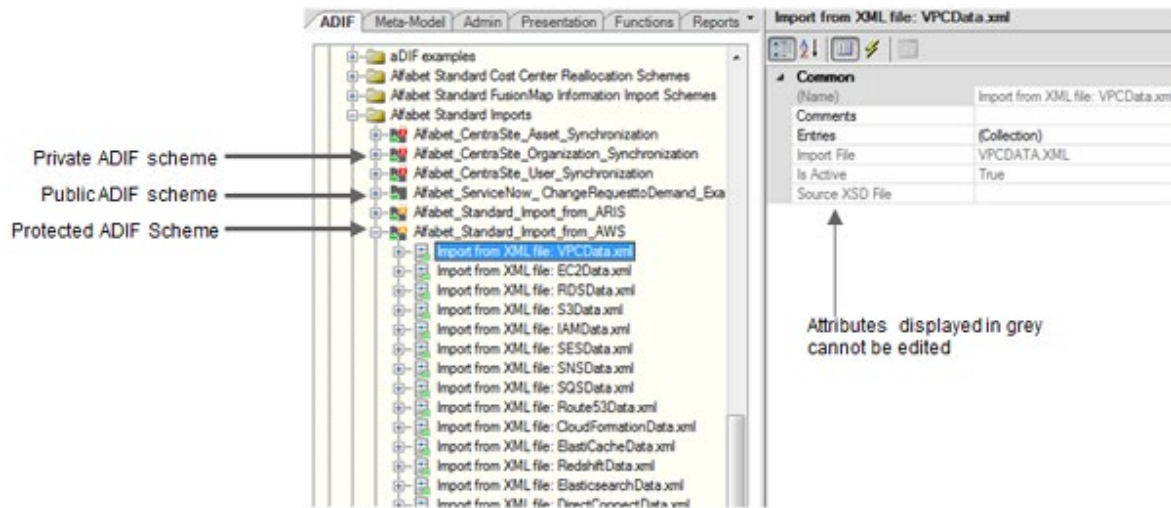
FIGURE: Private, protected and public ADIF schemes in the ADIF Explorer

Copying or Moving ADIF Scheme Elements

The context menu of the nodes in the explorer also includes a copy and paste functionality to reuse already configured elements in another context.

Instead of copying elements you can also move the elements to another location using cut and paste.

1) In the explorer, right-click the element that you want to re-use and select **Copy** to copy it to another element or **Cut** to move it to another element.

2) In the explorer, right-click the parent element that you want to add the copied element as child element to and select **Paste**. The element is copied to the new location with the name <old name>_1.

3) Optionally, click the added element in the explorer and change the **Name** attribute in the property window.

⚠ Please note that this behaviour does not apply to copying of SQL commands into other SQL commands. If you try to copy an SQL command into another SQL command, the current SQL command is overwritten instead of adding the copied SQL command as child element. To copy an SQL command into another SQL command, add a new subordinate SQL command to the target parent SQL command and paste the copied SQL command into the newly created subordinate SQL command.

Changing the Order of Child Elements of an ADIF Scheme Element

Elements of an ADIF scheme are executed in the configured order. You can change the order of execution of the child elements within a node of the ADIF configuration interface explorer by changing the order of nodes.

ⓘ The following exceptions to the general mechanism apply:

- The mechanism for changing the sort order is the same for all nodes with the exception of the child nodes in the sub-folders **Relations** and **Indexes** of ADIF import entries.

- The sort order of sub-folders for SQL commands or attribute definitions in an ADIF import or export entry cannot be changed.

To change the order of sub-nodes in an ADIF scheme, an import or export set, or of one of the folders for SQL commands:

1) Click the node for which you want to change the order of child elements.

2) In the attribute section, click the attribute that you want to sort and click the **Browse** ![...] button to open the editor that allows sorting. The following attributes allow sorting:

- ADIF import scheme and ADIF export scheme nodes: **Items** attribute

- Import set and export set nodes: **Entries** attribute.

- Attributes folder of import entries: **Attributes** attribute.

- **SQL Commands –** < **ActionType** > folder: **Sub Commands** attribute.

An editor opens containing a list of the child nodes.
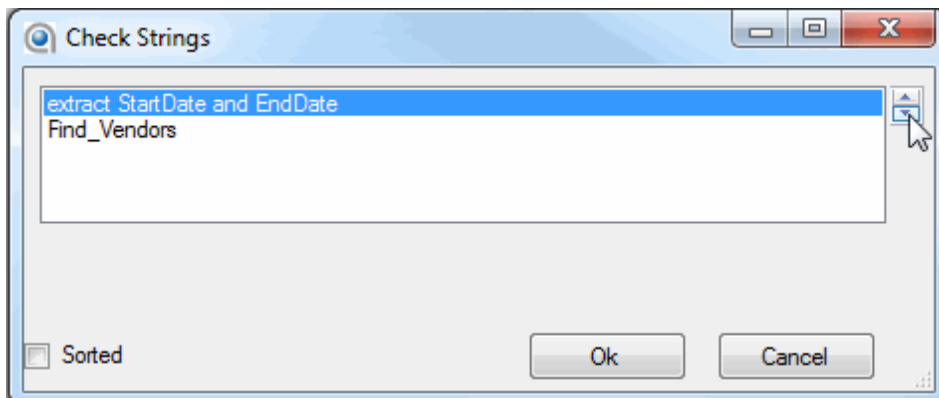
3) Click one of the child nodes in the list and click the **Up/Down** ![buttons] buttons in the upper-right corner to move the selected element up or down in the list. Sort all child nodes as needed.

> ⓘ Alternatively, you can select the **Sorted** checkbox in the lower-left corner to sort all elements alphabetically.

4) Click **OK** to save your changes. The order of child nodes in the explorer tree changes according to your settings.

To change the order of sub-nodes in the **Relations** or **Indexes** folder of an ADIF import entry:

1) In the explorer, expand the folder for which you want to change the sort order of child nodes.

2) Click the child node that you want to move within the folder and drag it to the new position.

Deleting ADIF Scheme Elements

You can delete either complete ADIF schemes or elements of ADIF schemes. If you delete a node from the explorer, all sub-nodes will also be deleted.

To delete an ADIF scheme:

1) In the explorer, right-click the ADIF scheme that you want to delete and select **Show Usage**. A new window opens that lists all event templates and buttons configured to trigger the execution of the ADIF scheme.

2) If the list displays any usage of the ADIF scheme, remove the configurations using the ADIF scheme prior to deleting the ADIF scheme.

3) In the explorer, right-click the ADIF scheme and select **Delete**. The element and all its sub-elements are deleted.

To delete an ADIF scheme element:

1) In the explorer, right-click the ADIF scheme element that you want to delete and select **Delete**.

Structuring ADIF Schemes in Groups

The ADIF configuration interface allows you to structure ADIF schemes in group nodes. These elements are not part of the ADIF XML specification but are empty explorer nodes used to visually structure the ADIF scheme definitions.

A group is created by adding at least one ADIF scheme to the group:

1) In the ADIF explorer, select one of the ADIF schemes that you want to add to the new group.

2) In the attribute window of the ADIF scheme, enter the name of the group to the attribute **Group**. A new group node with the specified name is created in the explorer as parent element of the selected ADIF scheme.

3) You can now do any of the following:

- To add another already existent ADIF scheme to the group, set the **Group** attribute of the ADIF scheme to the name of the existing group.

- To add a new ADIF scheme to the group, right-click the group node and select either **Create Import Scheme** or **Create Export Scheme**.

- To move an ADIF scheme to the root level of the hierarchy, delete the group name from the **Group** attribute.

- To move an ADIF scheme to another group, alter the group name in the **Group** attribute of the scheme. The target group can either be an existing group or a new name that leads to the creation of a new group in the explorer.

- To delete a group, alter the **Group** attribute of all ADIF schemes in the group. A group node exists as long as it is defined in one of the ADIF schemes.

Defining SQL Queries for SQL Commands

ADIF import and export schemes are mainly configured via SQL queries. The queries are defined in the **Text** attribute of SQL command elements within the ADIF scheme. If you click the **Browse** button in the field for the **Text** attribute, a text editor opens that allows you to define the query.

Defining SQL Queries for Different Database Servers in One SQL Command

The SQL syntax depends on the database server that the query is to be executed on. You might want to execute an ADIF scheme on databases located on either a Microsoft SQL Server or an Oracle database server. For this situation, you can define the query for an SQL command in different syntaxes within the same ADIF scheme. Each version of the defined query must start with the name of the server type in a comment (a separate line starting with two dashes and a whitespace).

Allowed server type definitions are:

- `SQLSERVER`

- `ORACLE`

> For example, a query defined for execution on either a Microsoft SQL Server or an Oracle database server:
>
> ```
> -- SQLSERVER
>
> UPDATE TMP_TABLE SET TMP_TABLE.ATTR1 = APPLICATION.ATTR2 FROM
> APPLICATION WHERE APPLICATION.REFSTR = TMP_TABLE.REFSTR
>
> -- ORACLE
>
> UPDATE TMP_TABLE SET (TMP_TABLE.ATTR1, TMP_TABLE.ATTR2) = (SELECT
> ATTR1, ATTR2 FROM APPLICATION WHERE TMP_TABLE.REFSTR =
> APPLICATION.REFTR)
> ```

Help for Defining SQL Queries

The text editor for the definition of SQL queries in the ADIF configuration interface offers help for the definition of the SQL query in separate tabs:

- References to other object classes

When you first open the text editor, a tab **AdminDesktop** is displayed. The object class `AdminDesktop` is the first class in alphabetical order in the Alfabet meta-model. In the **Select Class** field, select the object class whose references to other object classes you want to see. A new tab will open with the name of the selected object class. The tab displays a graphic showing the relationships between the selected object class and other object classes.

The selected object is displayed in the centre of the graphic. On the left, all object classes are listed that reference the selected object class via one or multiple of their properties. The property that establishes the reference is displayed in a white box between the two classes. On the right, all properties of the selected class are displayed in a white box that store references to other object classes. The target classes are listed in the blue boxes on the right.

Object classes with a reference to Assignment (blue) and the properties establishing the reference (white)

Properties of the object class Assignment (white) that reference a target object class and the target object classes (blue)

When you double-click any referenced object class in the graphic, a new tab will open that displays the references of the selected object class to other object classes.

- Object class documentation

When you open the **[text]** tab and select an object class in the **Select Class** field, the help document provided for the object class is displayed. The documentation includes a description of the purpose of the object class and of all the properties of the object class and lists all relevant attributes in a table. Additionally, information is provided about class dependencies based on database triggers. The documentation also includes the rules that define which dependent objects are deleted when an object of the selected class is deleted.

The Meta-Model Sub-Tree of the ADIF Explorer

The **Meta-Model** sub-tree of the ADIF explorer provides information about the Alfabet meta-model that may be useful during configuration of the ADIF interface. The **Meta-Model** explorer nodes has two sub-nodes: **ENUMERATIONS** and **Class Model**.

ENUMERATIONS

The **ENUMERATIONS** sub-tree lists all currently configured enumerations. If a property is based on an enumeration, you can check which input values are permissible for the properties by clicking on the enumeration in the explorer and clicking the **Browse** button for the **Items** attribute. A text editor opens that lists each permissible value in a separate row.
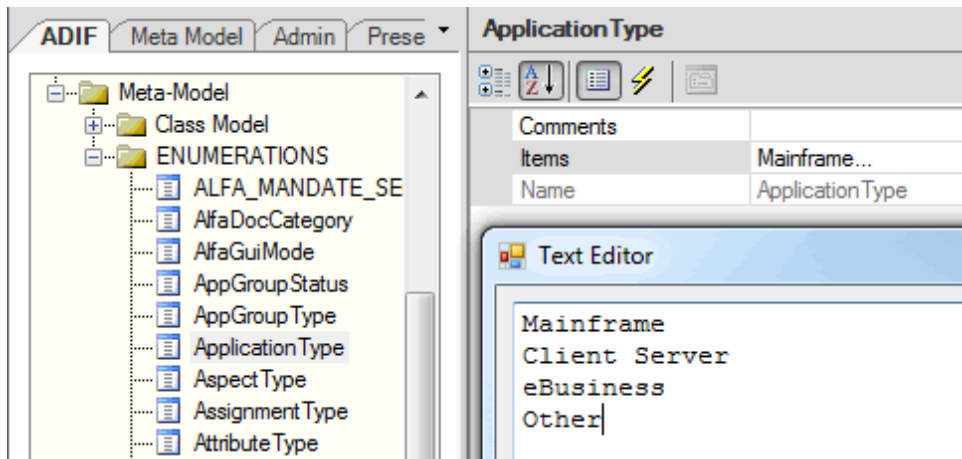
FIGURE: Permissible values for properties based on the enumeration ApplicationType

Class Model

The **Class Model** sub-tree lists all classes of the Alfabet meta-model. The classes are structured according to their use in the application. The classes listed in the **SEMANTICCLASS** folder are relevant for ADIF. The **SEMANTICCLASS** folder has the following sub-folders defined:

- **ALFA_QUESTIONARY**: Customer defined, temporary classes that are relevant for data collection via the Questionnaire capability.

- **Artifact**: Classes used to store data about the customer's IT infrastructure. All **Artifact** classes that can have a responsible user defined are listed in the sub-folder **ArtifactAuthorized** of the **Artifact** folder.

- **ITClass**: Object classes supporting functionality. This includes, for example, the class `TimeStatus` that is used to store the lifecycle of objects or the class `RoleType` that is used to store information about which roles a user can have with regard to an object.
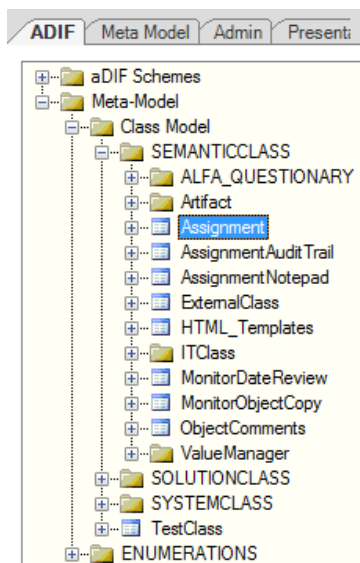
FIGURE: Class Model explorer with the folder SEMANTICCLASS expanded

The following information about each object class is available via the **Class Model** explorer:

- Object class attributes

When you click an object class in the explorer, the attributes of the object class are listed in the attribute section on the right of the explorer.

⚠ Please note that object classes having the attribute **Automatically Managed** set to true are object classes that shall only be changed by mechanisms triggered by the Alfabet software components and must not be changed by ADIF import or any third party component.

- Object class properties

When you expand an object class node in the explorer, a folder **PROPERTIES** is displayed. When you expand the **PROPERTIES** node, you can see the properties of the object class listed in the explorer. Mandatory properties are highlighted in yellow and written in red letters. When you click a property node, the attributes of the property are listed in the attribute section to the right of the explorer.

ⓘ The property Name of the object classes listed under the **Artifact** node is inherited from the **Artifact** parent class. It is marked as mandatory because it is mandatory for most of the sub-ordinate object classes. The sub-ordinate classes inherit all attributes of the parent class property. Therefore, the Name attribute is also marked as mandatory for the few subordinate classes that do not require a Name attribute.

- References to other object classes

When you right-click an object class in the explorer and select **Show Class Navigator**, a new window opens between the explorer pane and the attribute window. The relation of the selected object class to other object classes via references is displayed in a graphic.

The selected object is displayed in the centre of the graphic. On the left, all object classes are listed that reference the selected object class via one or multiple of their properties. The property that establishes the reference is displayed in a white box between the two classes. On the right, all properties of the selected class are displayed in a white box that store references to other object classes. The target classes are listed in the blue boxes on the right.

Object classes with a reference to Assignment (blue) and the properties establishing the reference (white)

Properties of the object class Assignment (white) that reference a target object class and the target object classes (blue)
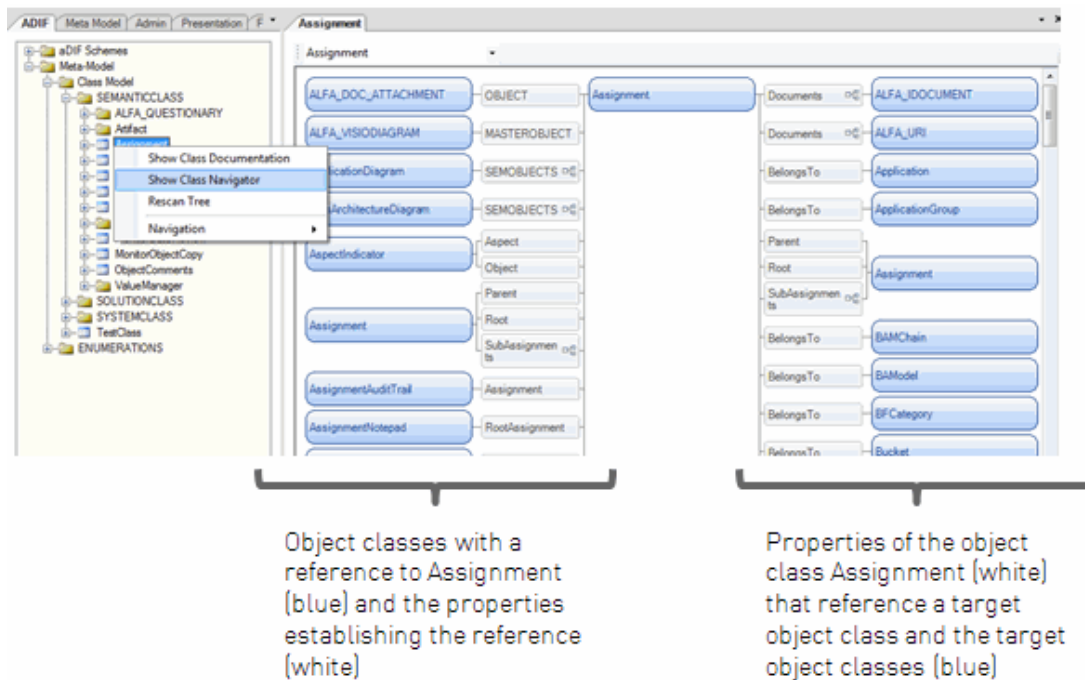
- Object class documentation

When you right-click an object class in the explorer and select **Show Class Documentation**, a new window opens between the explorer pane and the attribute window. The help document provided for the class is displayed. The documentation includes a description of the purpose of the object class and of all the properties of the object class and lists all relevant attributes in a table. Additionally, information is provided about class dependencies based on database triggers. The documentation also includes the rules that define which dependent objects are deleted when an object of the selected class is deleted.

Configuring ADIF via XML Editing

The configuration of the ADIF scheme is XML based. Instead of configuring ADIF schemes in the ADIF or Alfabet Expand user interface, the schemes can be created or edited in an XML editor. The ADIF explorer tree offers a means to open the scheme in an internal editor that provides help and debugging functionality. Alternatively, you can export any scheme and edit it in any standard XML editor or create a scheme from scratch in an XML editor and import it into the ADIF user interface.

Internal XML Editor

To open the ADIF scheme XML object in the internal XML editor of the ADIF user interface:

1) In the ADIF explorer, right-click an ADIF scheme and select **Open XML Editor**. The XML editor opens in a new window.

The ADIF internal XML editor provides various help functionalities for the creation of an ADIF scheme:

- Help for data entry:

When you write a < into the editor, a list of elements that are allowed at this position of the XML code is displayed. You can double-click an element name in the list to add it to the code. When you add a whitespace after an element name, a list of attributes allowed for the element are displayed. You can add an attribute by double-clicking the attribute in the list.

- Hierarchy information for elements and attributes:

If you move the cursor over an element name, the parent hierarchy for the element will be displayed. If you move the cursor over an attribute name, the name of the element that the attribute is defined for will be displayed.

- Error messages in case of incorrect or incomplete elements:

Error messages are displayed below the edit pane when you add an element that is not allowed in the XML schema or at the position you added to, or when attributes or an attribute value do not correspond to the schema specification. The error messages provide information about the line and column position of the error and about the kind of error that has occurred.

External Editors

You can create and edit ADIF schemes in any text editor.

The edited ADIF schemes can either be uploaded to the database via the ADIF configuration user interface or provided via the local file system. The ADIF console application used to start the import or export processes works with both an ADIF scheme provided via XML file or with an ADIF scheme that is part of the target/source database.

You can export a single ADIF scheme configured via the ADIF configuration interface to an XML file for editing in an external XML editor.

To save ADIF schemes in an XML file.:

1) In the ADIF explorer, right-click the ADIF scheme that you want to export.

2) In the context menu, select **Save as**.

3) In the explorer that opens, enter a name for the XML file and store it at the desired location.

Completed ADIF schemes that are configured in an XML file can be uploaded to the database as follows:

1) In the ADIF explorer, right-click the root node **ADIF Schemes**.

2) In the context menu, select one of the following:

   - **Merge from Files**: New schemes are added to the database and schemes that are available in both the XML file and the database are overwritten in the database with the ADIF schemes in the file. Schemes that are only available in the database are not altered. During the merge action, the ADIF schemes are mapped according to their **Name** attribute.

   - **Replace from File**: All ADIF schemes in the database are deleted and the ADIF schemes in the XML file are then uploaded to the database.

3) Select the file in the explorer that opens. For the **Merge from Files** option multiple files located in the same folder can be selected. For the **Replace from File** option only one file can be selected.

Chapter 4:   Configuring Data Import with ADIF

This chapter guides you through the process of importing data to the Alfabet database via the ADIF interface.

Data Processing During Import

Data import can include data from database tables, Microsoft® Excel® files, comma-separated data format files (.csv or.txt), JSON files, and XML files. All files included in the import must be provided in a ZIP file. Import can be performed as single import process based on multiple import sources.

> ℹ️ ZIP files are checked during upload for the size of content after decompression. The file will not be uploaded if the decompressed size is more than 100 percent of the compressed size or if storing the file content on the local drive would result in less than 1 GB free space remaining or if any deviations from normal compression mechanisms are detected.

During import, the external data must be mapped to the Alfabet database:

- The external data table must be mapped to the correct database table in the Alfabet database.

- The external data values must be mapped to the correct columns in the Alfabet database table.

- The data records in the external data table must be mapped to the correct rows in the existing Alfabet database table. In most cases, the Alfabet database already contains data. In this case. The external data should not simply be added to the Alfabet database table. Object data should be updated if the object already exists and new records should only be written to the database for new objects. It may even be necessary to delete existing data during import.

The ADIF import mechanism targets the structure and uniqueness constraints of the Alfabet database. The information about the external data must be specified by the customer to allow the ADIF import mechanisms to map the data. Customers can configure the import via SQL.

The structure of the external data can vary considerably from the structure of the Alfabet database tables. In order to keep the import process as flexible as possible, the ADIF import mechanism performs the import in two steps:

- Step 1: Temporary tables are built based on the data in the input file / external database table. The temporary database table can be altered and completed via SQL commands before beginning the second import step.

- Step 2: Data from the temporary tables is written to the Alfabet database tables for object classes in the Alfabet meta-model.

This mechanism ensures a high degree of flexibility. You can define multiple temporary tables with a different structure and different content from your external data (and even define temporary tables from the content of already generated temporary tables) in order to restructure the data according to your needs. The Alfabet database tables are then altered on the basis of the content of the temporary tables rather than the content of the original external data.

After the import is complete, the temporary tables will be deleted from the Alfabet database.

Execution Steps of the Import Process

Import is triggered by the ADIF import schemes that have been configured by the customer via SQL commands. An ADIF import scheme can trigger import from multiple database tables or files of one or more file formats.

The import process is started via the ADIF debugger or via the ADIF console application.

The following process is executed when the ADIF import is started:

1) If the data is imported from one or more files, the import files (which must be located in a ZIP file) will be uploaded to the Alfabet database.

2) Validation of the ADIF import scheme is performed. Import will be stopped if validation fails.

3) A temporary directory is created on the server side for log files and import files.

4) The ZIP file with import files is uploaded to the temporary directory and unzipped.

5) Command line arguments are written to the Alfabet database. If the import is triggered by an ADIF import scheme located in an XML file, the ADIF import scheme is also written to the database.

6) Temporary tables specified in all import specifications in the ADIF import scheme are written to the database. Details are provided in this chapter.

7) The Alfabet database tables are updated with data from the temporary database tables as defined in the import specifications in the ADIF import scheme. Details are provided in this chapter.

8) All specified workflows are started.

9) Log information about the import process is uploaded to the `ADIF_SYS` folder in the **Document Explorer** in the Alfabet database.

10) The temporary directory on the server side (that was created prior to import execution) and all content in the temporary directory is deleted.

11) If not otherwise specified in the ADIF import scheme, all temporary tables created during import will be dropped.

12) If import is started via the ADIF console application, the log file content is downloaded to the console log file.

The following section describes how to configure different import features within the ADIF import scheme. Details about the processes executed to perform the import steps described above and the order of execution of different ADIF scheme elements processed during import is provided with the description of the various import steps.

Preparing Data For Import

Data can be imported from the following data formats:

- XML files

- JSON files

- Microsoft® Excel files (.xls and.xlsx)

- CSV files

- external database tables

ADIF is highly flexible concerning the translation of external data structures to fit the Alfabet meta-model. Nevertheless, to ensure that data can be read properly during the import process, the external data that is used for import must adhere to the following requirements.

The following sections list the specific requirements for all the individual import file formats.

Microsoft® Excel® Files

Import can be performed from.xls and.xlsx files. The following rules apply to both file formats.

- All data must be available in the first tab of the Microsoft® Excel® file. All other tabs are ignored during import.

- If the data contains a header, the header cells must only contain alphanumeric data without any special characters.

- The table must have a flat structure.

- The allowed data formats are numbers, strings and dates. Any special characters, including NULL values (hexadecimal value 0x00) cannot be imported.

- The table must not contain formulas.

XML Files

- The file must contain valid XML definitions.

- The following encoding is allowed:

  - ASCII

  - UTF7

  - UTF8

  - UTF32

  - Unicode

- Text content of XML elements can only be interpreted by the import mechanism when the XML element containing the text has no own attributes and is a child element of an XML element containing attributes.

- If the XML file refers to a DTD file, the DTD file must be included in the import ZIP file when importing the XML file.

- Strings containing line breaks must be written in CDATA elements or the line breaks must be escaped according to XML rules.

- Data has to be XML compliant to be imported. All characters that are not XML compliant will be replaced automatically with whitespaces during import.

JSON Files

- The file extension must be .json.

Comma-Separated Formats (*.csv)

Import can be performed from.csv and.txt files. The following rules apply to both file formats.

- The data must represent a table with each row written in a new line. Each column within a row must be separated with a column separator. Each data record (row) must have the same number of columns.

- The following encoding is allowed:

  - ASCII

  - UTF7

  - UTF8

  - UTF32

  - Unicode

- The file size must not exceed the maximum number of datasets that can be processed with Microsoft® Excel® in a single data sheet. The same component is used for importing data from Microsoft Excel and CSV files and this component has an integral processing limit that corresponds to the limits defined for Microsoft Excel.

External Databases

Import can be performed from the following types of databases:

- Microsoft® SQL

- Oracle®

- Microsoft® Access

- PostgreSQL 9.1

- LDAP tables

You must ensure that access permissions of the external database allow the data to be read by the ADIF process.

A connection string for connection to the database must be added to the configuration and should be provided prior to configuration.

Conceptualizing Data Import

The first and most important step for data import via ADIF is the conceptualization of the data import. You should clarify the following before configuring the ADIF import scheme:

- Which data shall be imported?

- How can this data be mapped to the Alfabet meta-model?

- Is configuration of the Alfabet meta-model required? For example, are custom properties required for object classes because the data that shall be imported is not represented by the standard properties of the Alfabet meta-model?

- What are the data formats of the import data and do they match the data formats of the Alfabet database tables?

- Are all mandatory properties for the Alfabet object class included in the import of new data records?

- Does my data include unique attributes that allow the data to be mapped to existing Alfabet database tables?

- Do my data import formats adhere to the rules described in the section *Preparing Data For Import*?

- If the import data includes relations to other objects, do the database table for the related object classes require update prior to import?

- What intermittent steps using temporary tables are necessary to import the data successfully to the Alfabet database?

It is recommended that you provide a thorough concept in written form to the person configuring the ADIF import scheme.

Configuring the ADIF Import Scheme

An ADIF import scheme is an XML object that can be edited either in an XML editor or in the ADIF configuration interface provided as part of the configuration tool Alfabet Expand. In the interface, the XML elements are displayed as nodes in the explorer, while the attributes of an element are displayed in the attribute window to the right of the explorer.

All information and commands required to import data from the Alfabet database are defined via the elements of the ADIF import scheme. The elements must follow a defined sequence that is given by an XSD scheme. In the ADIF explorer, the context menu options available for the import scheme elements guide you through the creation of all required elements. For example, the context menu displays only the options to create permissible sub-elements for an import scheme element. When you define the ADIF import scheme in an XML editor, you must consult the XSD scheme for information about the permissible sequence of objects.

The following elements are part of the ADIF import scheme:

XML Element Name	Caption in the ADIF Interface	Purpose
`ADIF_Import Scheme`	Import Scheme	Element for the configuration of basic overall import execution parameters.
`Parameter`	Parameter	Element defining a parameter definition that may be used in all queries within the ADIF import scheme as variable that is filled with a value defined when executing the ADIF import. This element is not visible in ADIF import schemes defined prior to Alfabet 10.4 for that the **Parameters Backward Compatibility Mode** attribute is set to `True`.
`XmlImportSet`	XML Import Set / JSON Import Set	Element defining a JSON or XML file structure for data import from XML files. ⓘ For import from JSON files, the option to create a JSON Import set is available for backwards compatibility reasons only. For import from JSON, an entry for hierarchical JSON should be created instead.
`FileImportSet`	File Import Set	Structuring element that is a container for elements defining the import from Microsoft® Excel® files or CSV files.
`DBImportSet`	DB Import Set	Element defining the connection to an external database.
`ADImportSet`	LDAP Import Set	Element defining the connection to an external LDAP table.
`ImportEntry`	Entry	Element defining the import from one external database table or file to the Alfabet database.
`Attribute`	Attribute	Sub-element of `ImportEntry` elements for the definition of the mapping of external data to a database table column in the Alfabet database.
`Relation`	Relation	Sub-element of `ImportEntry` elements for the definition of a relation between objects that shall be created during import.
`Index`	Index	Sub-element of `ImportEntry` elements for the definition of an index to be created during import.

XML Element Name	Caption in the ADIF Interface	Purpose
`SQLCommand`	SQL Command	Sub-element of `ImportEntry` elements for the definition of SQL commands to be executed during import.
`AlfaMethod`		Sub-element of `ImportEntry` elements to assign custom code to the entry. `AlfaMethod` elements are only relevant if custom code was developed by Software AG for special customer requirements. Information about the configuration is provided individually with the custom code.

Creating an ADIF Import Scheme

The following describes the creation and editing of an import scheme via the ADIF configuration interface. Please note that only the set of the attributes required for the task that is being described are explained.

To create a new ADIF import scheme:

1) In the explorer, right-click the **ADIF Schemes** root node and select **Create Import Scheme**. The new import scheme is added to the explorer. The attribute window of the new import scheme is displayed on the right.

2) In the attribute window, set the following attributes for the ADIF import scheme:

   - **Name**: Enter a unique name. The name is used to identify the ADIF import scheme in technical processes. It must be unique and should not contain white spaces or special characters.

   - **Caption**: Enter a meaningful and unique caption. The caption is used to identify the ADIF import scheme in the Alfabet user interface in the **ADIF Jobs Administration** and **My ADIF Jobs** functionalities.

   - **Description**: Enter a meaningful and short description of the result of the ADIF import. The description is displayed in the Alfabet user interface in the **ADIF Jobs Administration** and **My ADIF Jobs** functionalities in the preview window for an ADIF scheme.

   - **Commit After Run** : If set to `True`, the result of the data import is written persistently to the Alfabet database. If set to `False`, the import process will be rolled back after execution and no changes will be written to the database. Configuration of the automatic start of workflows during import is ignored when **Commit After Run** is set to `False`. It is recommended that you set **Commit After Run** to `False` for a new import scheme to allow debugging without the risk of corrupting the database. After the successful testing of the data import and verification that the resulting changes to the Alfabet database are as expected, you can reset the **Commit After Run** attribute to `True` to perform regular data import.

     ⚠ Please note the following:

       - Setting the **Commit After Run** attribute rolls back all changes to data records in existing tables caused by DML statements. The creation or deletion of tables is not included in the roll back. For example, if you test an ADIF

scheme that is configured to persistently write temporary tables to the database, these temporary tables will be created persistently even if **Commit After Run** is set to `False`. SQL commands of the type **OnActivate** are also excluded from roll back.

- When new objects are created during an ADIF import job, the data bind mechanism assigns `REFSTR` values for the new objects. When **Commit After Run** is set to `False`, the objects are not created in the database, but nevertheless the `REFSTR` values are regarded as in use and will not be used for data bind in the next ADIF run unless the Alfabet Server or Alfabet Expand application used to process the ADIF jobs is restarted.

- Changes triggered by **OnActivate** commands are not rolled back if the option **Commit After Run** is set to `False` for the import scheme.

- **Drop Temp Tables** : If set to `True`, all temporary tables are dropped after import. Only the changes to the Alfabet database are stored persistently. If set to `False`, the temporary tables are kept in the database after import is finished. Storing temporary tables persistently is only required for special import/export cycles designed for data manipulation that require input from the temporary tables of a previously set import. In most cases, setting this attribute to `True` is recommended to clean the database of data that is not part of the Alfabet meta-model.

- **File Post Action:** This attribute defines the handling of import files after import. Select one of the following:

  - `None`: Import files are not altered.

  - `Delete`: Import files are deleted after import. If import is based on a *.zip file, the *.zip file is deleted. If files are read from a directory, all files in the specified import directory will be deleted.

  - `RenameUsingTimeStamp`: Import files are renamed after import. If import is based on a *.zip file, the *.zip file is renamed. If the import reads files from a directory, all files in the specified import directory are renamed. The renamed files will have a timestamp added to the file name and all letters of the file name will be changed to capital letters.

    The `RenameUsingTimeStamp` parameter is ignored during debugging.

- **Import File Required** : Select `True` if your import scheme includes configurations for data import from files. Set this parameter to `False` to import data from external databases or Active Directories only or when defining import based on an assistant. Assistants are available for pre-defining data import from external services in the scope of Alfabet integration solutions. For more information, see Predefined ADIF Schemes.

- **Block Processing**: If a new object is created via the ADIF import and a default value is defined for properties of the object class, the import mechanism checks whether the import includes a value for the property. If no value is provided, the default value will be set. To enhance performance and processing time of ADIF import jobs for very large-scale datasets (in excess of 1 million records) in batch, the **Block Processing** attribute of the ADIF scheme can be set to `True`. The block processing mechanism scans all object class properties for that no column is defined in the temporary table of the import. If a default value is defined for the object class property, an additional column will be added to the temporary table for this object class property, returning the default value for each row. This will spare the import mechanism to

set the default value for each object separately. In addition, **Block Processing** improves performance for rescanning back relations for imports that are including import of relations with corresponding back relations in the target objects. **Block Processing** should only be selected if the import dataset is very large and a number of object class properties for that no value is provided via the imported data have a default value definition.

- **Auto Run**: If set to `True`, the ADIF scheme will be executed automatically after each update of the Alfabet meta-model via an ***.amm** file or each restore of the database with an ***.adb** file.

> Note the following:
>
> - If the **Auto Run** attribute is set to `True`, the **Import File Required** attribute must be set to `False`.
>
> - If ADIF import schemes are executed automatically, no log files will be generated. The ADIF scheme must be tested during configuration to ensure that it is properly executed at runtime.
>
> - The database connection is closed and re-opened before execution of an auto-run. This may take some time.

> For more information about the update of the meta-model and the restore of the database, see the reference manual *System Administration*.

- **Alfabet User Interface Behaviour**: Select one of the following to define the availability of the ADIF scheme in the Alfabet user interface:

  - `VisibleExecutable`: The ADIF import can be triggered in the **ADIF Job Administration** functionality and the success of the ADIF jobs executed with this ADIF scheme can be controlled via the **ADIF Job Administration** and **My ADIF Jobs** functionalities.

  - `VisibleNotExecutable`: The success of the ADIF jobs executed with this ADIF scheme can be controlled via the **ADIF Job Administration** and **My ADIF Jobs** functionalities, but the ADIF import cannot be triggered via the Alfabet user interface.

  - `NotVisible`: The ADIF scheme and the information about ADIF jobs executed with this ADIF scheme are not visible in the **ADIF Job Administration** and **My ADIF Jobs** functionalities. this is the default value for new ADIF import schemes.

  > For information about administration and execution of the ADIF scheme via the **ADIF Job Administration** and **My ADIF Jobs** functionalities, see Executing and Controlling ADIF via the ADIF Jobs Administration and My ADIF Jobs functionalities in the Alfabet User Interface.

- **Applicable for REST API**: Set the attribute to **True** if the ADIF scheme shall be executed via a RESTful service call to the endpoint `adifimport` of the Alfabet RESTful API either via a RESTful service call from an external RESTful client or via an Alfabet event that is triggering the execution of the ADIF scheme when a user enters or leaves a wizard or workflow step or when an event for execution of a RESTful service call to a third party application is finished.

  > Execution of the ADIF scheme via the Alfabet RESTful API requires setup of the RESTful services. For information about the requirements and the execution of the service call, see the reference manual *Alfabet RESTful API*.

For the additional configuration to implement execution of an event, see *Configuring Events*.

- **Group**: The ADIF configuration interface allows you to structure ADIF schemes in group folder nodes. These elements are not part of the ADIF XML specification but are empty explorer nodes used to visually structure the ADIF scheme definitions in the explorer. Enter a name of an existing group folder to add the ADIF scheme to the group or enter a new name to create a new group folder node and add the ADIF scheme to the new group.

After creating the ADIF import scheme, you must now add import definitions to trigger import from one or more import data formats to the import scheme. Proceed to the section *Configuring Import from Different External Source Formats* to configure the import scheme.

Optionally, you can configure the following:

- *Configuring Execution of the Import Scheme Dependent on Current Parameters*

- *Configuring SQL Commands for Optional Enhanced Import Features*

- *Configuring the Import Audit History*

- *Configuring the Automatic Start of Workflows During Import*

- *Configuring Logging Parameters*

Configuring Import from Different External Source Formats

The import of data to a temporary table prior to the update of the Alfabet database tables divides the configuration issues in two parts. The configuration of how to import data to the temporary tables (which depends on the different import formats) and the definition of mapping conditions to update data in the Alfabet database tables (which is identical for all import formats).

For each import, you must specify the following:

ADIF Scheme Element	For Import from Database Tables	For Import from Files (XLS, XLSX, CSV, TXT)	For Import from XML	For Import from JSON
Import Set	The connection to the external database must be defined in a database import set or an LDAP import set The import set is parent to all other configuration elements for the import from that database.	The specification of an import set is not required. Nevertheless, you can define file import sets as structuring elements containing, for example, all import definitions for import from the same file format or for import regarding the same task.	For each external XML file, an XML import set must be added to the import scheme. The import set is parent to all import definitions for XML elements in the file.	The specification of an import set is not required. For backward compatibility reasons it is still possible to create an import set for JSON import. This import did not cover all use cases and was therefore substituted by a direct JSON import from an import entry.
Import Entry	At least one import entry must be added to the database import set for each external database table from which data shall be imported. An import entry specifies import between one external database table and one Alfabet database table only. If you want to import data	At least one import entry must be added to a file import set or directly to the ADIF import scheme for each file from which data shall be imported. An import entry specifies import between one external file and one Alfabet database table only.	At least one import entry must be added to the XML import set for each XML element type in the XML file that data shall be imported to. An import entry specifies the import between one XML element type and one Alfabet database table only. If you want to import data from one	At least one import entry must be added to the ADIF import scheme via the context menu option **Create Entry for Hierarchical JSON** for each file from which data shall be imported. The resulting import entry has a predefined temporary table structure.

ADIF Scheme Element	For Import from Database Tables	For Import from Files (XLS, XLSX, CSV, TXT)	For Import from XML	For Import from JSON
	from one external database table to multiple Alfabet database tables, you must specify multiple import entries.	If you want to import data from one external file to multiple Alfabet database tables, you must specify multiple import entries.	XML element type to multiple Alfabet database tables, you must specify multiple import entries.	An import entry specifies import between one external file and one Alfabet database table only. If you want to import data from one external file to multiple Alfabet database tables, you must specify multiple import entries.
Attribute Relation Index Indicator Role SQL Command	The child elements of the import entry map the content of the external database table, the columns of the external Microsoft® Excel® table, and the comma- separated values in the comma-separated external file. Various types of child elements are available to ease the mapping of data to Alfabet database columns, the creation of relations between objects, or the creation of an index. Mechanisms provided via the ADIF scheme or the specification of SQL commands provide support for mapping records to already existing records of the database tables.			

To learn how external data is interpreted during import and how the import set must be defined, proceed to the section relevant for your import format:

- *Defining Data Upload from an External Database in a Database Import Set*

- *Defining Data Upload from an LDAP Table in an LDAP Import Set*

- *Defining Data Import From XML Files in an XML Import Set*

- *Defining Data Import from Microsoft® Excel Files and Comma-Separated File Formats*

Defining Data Upload from an External Database in a Database Import Set

If you want to import data from an external database, a connection to the external database must be established to retrieve the data. The ADIF import process does not read the data directly from the database tables of the external database. Data must be uploaded via a customer-defined query executed on the external database. The data set resulting from the query is the external data imported via the ADIF import process.
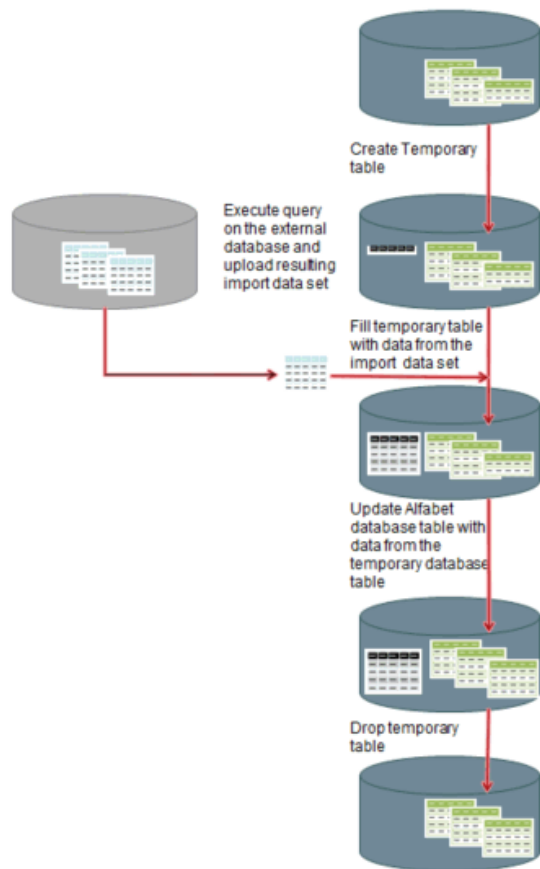
FIGURE: Import of data from an external database

To configure data upload from the external database:

1) In the explorer of the ADIF configuration interface, right-click the ADIF import scheme and select **Create DB Import Set**. A new database import set ⬛ is added to the explorer with a default name for the database import set. Below the database import set ⬛, a folder is automatically added for the creation of SQL commands that will be executed on activation of the database import set.
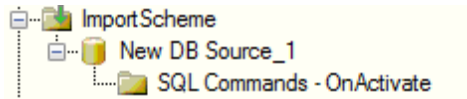
2) In the attribute window of the database import set "", specify the following attributes to configure the connection to the external database:

- **Name**: Enter a meaningful name for the database import set.

- **Comments**: Enter a comment that provides information about the functionality implemented within the database import set.

- **Driver Type**: Import can be performed from databases on Microsoft® SQL Servers®, Oracle® Database servers or Microsoft® Access databases. Select the type of database server that the external database is located on:

  - `SqlServer` for access to Microsoft SQL Server

  - `Oracle` for access to Oracle

  - `Access` for access to Microsoft Access via an OLE driver

  - `ODBC` for access to Microsoft Access, Oracle® MySQL or PostgreSQL 9.1 via an ODBC driver

  - `Hadoop` for access to Hadoop systems

- **Driver Sub-Type** : This attribute is only visible if `SqlServer` is selected in the **Driver Type** attribute. Select the driver that shall be used for the connection to the Microsoft SQL Server:

  - `MSNetServer`: The driver used in previous Alfabet releases. It is restricted to use with.NET Framework. This is the default value.

  - `MSSqlServer`: A driver supporting both.NET Framework and.NET Core environments.

- **Connection String**: Enter the connection string required to connect to the external source. The connection string depends on the database server used. It typically contains source location, user name, and other parameters relevant to the requirements of the database server.

  ⚠️ It is recommended that you define the database login in the connection string via a user name and password. If Windows Authentication is used for login and the ADIF Console Application is started with a remote alias, the connection to the database will only be successful if the Alfabet Server is started with the same domain user name and access rights that are provided with the connection string.

- Server variables can be used in the connection string. Server variables allow you to define all or part of the definition in the server alias configuration of the Alfabet Server instead of directly defining it in the ADIF scheme. Use of server variables is useful, for example, when using the configuration in a test and production environment with different external sources. The same external source definition can be used in both environments. The server alias definition used in the test and production environment define the correct connection data for the external source used in the respective environment. The use of server variables is described in the section *Using Server Variables in Web Link and Database Server-Related Specifications* in the chapter *Configuring Reports* in the reference manual *Configuring Alfabet with Alfabet Expand*.

- Command line parameters can be used as variables in the connection string.

ODBC can be used for connections to Microsoft Access or PostgreSQL 9.1 database servers. Please note, however, that performance will be lower compared to native drivers and stored procedures are not supported.

The following are examples of connection strings for connections to different database servers.

Connection string for connection to a Microsoft SQL Server® with standard login:

```
Server=myServerAddress;Database=myDataBase;User Id=myUsername;Password=myPassword;
```

Connection string for direct connection to an Oracle® database server:

```
User Id=APP_ID;Password=*********;Direct=true;Data Source=servername;Port=Port;SID=orcl
```

Connection string for direct connection to an Oracle® databse server including the service name:

```
User Id=********;Password=*********;Server=ServerFQDN/IP;Unicode=True;Connection Timeout=60;Max Pool
Size=250;Direct=True;Service Name=ServiceName;Port=1521
```

Connection string for indirect connection to an Oracle® database server:

```
User Id=***;Password=*****;Server=orcl;Unicode=True;Connection Timeout=60;Max Pool Size=250
```

- **Is Active** : Select `True` to activate the execution of all import entries within the import set. Select `False` to deactivate the execution.

3) Right-click the database import set ⬛ in the explorer and select **Create Entry**. A new import entry ⬛ is added to the database import set.

4) Right-click the **SQL Commands - DataUpload** folder below the import entry and select **Create SQL Command**. A new SQL command is added to the folder.

5) Select the new SQL command in the explorer and set the following attributes in the attribute window:

- **Name**: Enter a meaningful name for the SQL command.

- **Comments**: Enter a comment that provides information about the functionality implemented by the SQL command.

- **Command Type** : Select `DMLStatement` in the drop-down list.

- **Result Type** : Select `Undefined`.

- **Ignore Errors** : Select `True` if you want the upload to be executed even if the SQL statement specified with the attribute **Text** results in an exception. Select `False` if you do not want the upload to be executed in case of an error in the SQL statement.

- **Is Active** : Select `True` to activate the SQL command. Select `False` to deactivate the SQL command.

- **Text** : Define an SQL query with a `SELECT` statement that returns a data set when executed on the external database. The resulting data set is uploaded by ADIF as import data. Either write the SQL query directly in the attribute field or click the **Browse** button to open an editor for the definition of the SQL query. Define the SQL query in the **SQL Text** tab of the editor.

  (i) The text editor for the definition of SQL queries in the ADIF configuration interface provides help for the definition of the SQL query in separate tabs. For more information, see *Defining SQL Queries for SQL Commands* in the section *Configuring ADIF Schemes*.

  (i) If you want to configure complex import scenarios, you can add additional SQL commands to the import entry that are executed prior to or after data upload. For more information about the definition of additional SQL commands within an import entry, see *Configuring SQL Commands for Optional Enhanced Import Features*.

6) Define the import of the data set created via the SQL command for data upload in the import entry. For information about the required configuration, see *Defining Import of External Data in an Import Entry*.

7) If more than one data set shall be uploaded from the external database, repeat steps 3.) to 6.).

After creating the database import set, you must now add import entries that map the data in the external database to the data in the Alfabet database. For more information, see *Defining Import of External Data in an Import Entry*.

Optionally, you can define SQL commands that allow you to deactivate this part of the import scheme if specified preconditions have been fulfilled. For more information, see *Configuring Execution of the Import Scheme Dependent on Current Parameters*.

Defining Data Upload from an LDAP Table in an LDAP Import Set

If you want to import data from an external LDAP table (for example, an Microsoft® Active Directory®), a connection to the LDAP table must be established to retrieve the data. The ADIF import process does not read the data directly from the LDAP tables. Instead, data must be uploaded via a customer-defined LDAP search filter executed on the LDAP table. The data set resulting from the search filter is the external data imported via the ADIF import process.
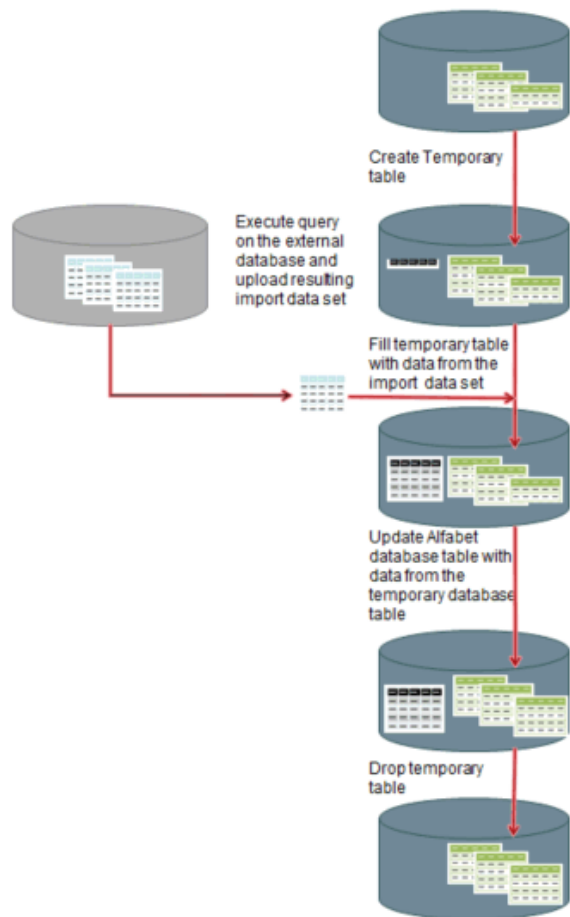
FIGURE: Import of data from an external LDAP table

To configure data upload from the LDAP table:

1) In the explorer of the ADIF configuration interface, right-click the ADIF import scheme and select **Create LDAP Import Set**. A new LDAP import set 🔲 is added to the explorer. A default name is provided for the LDAP import set. Below the LDAP import set 🔲, a folder is available for the creation of SQL commands to be executed on activation of the LDAP import set.

2) In the attribute window of the LDAP import set , set the following attributes to configure the connection to the LDAP table:

- **Name**: Enter a meaningful name for the LDAP import set.

- **Comments**: Enter a comment that provides information about the functionality implemented within the LDAP import set.

- **Authentication Method**: In the drop-down list, select the authentication method required for login to the external LDAP table:

  - `None`: Basic authentication (simple bind) is used in the LDAP provider.

  - `Secure`: Select if the WinNT provider uses NTLM to authenticate the client. Active Directory Domain Services uses Kerberos and possibly NTMLM to authenticate the client. When the user name and password are a null reference (Nothing in Visual Basic), ADSI binds to the object using the security context of the calling thread. This is either the security context of the user account under which the application runs or of the client user account that the calling thread impersonates.

  - `Encryption`: Attaches a cryptographic signature to the message that both identifies the sender and ensures that the message has not been modified in transit.

  - `SecureSocketsLayer`: Attaches a cryptographic signature to the message that both identifies the sender and ensures that the message has not been modified in transit. Active Directory Domain Services require the Certificate Server to be installed to support Secure Sockets Layer (SSL) encryption.

  - `Anonymous`: No authentication is performed.

  - `FastBind`: Specifies that ADSI will not attempt to query the Active Directory Domain Services objectClass property. Therefore, only the base interfaces that are supported by all ADSI objects will be exposed. Other interfaces that the object supports will not be available. A user can use this option to boost the performance in a series of object manipulations that involve only methods of the base interfaces. However, ADSI does not verify if any of the request objects actually exist on the server.

  - `Signing`: Verifies data integrity to ensure that the data received is the same as the data sent. The System.DirectroyServices.AuthenticationTypes.Secure flag must also be set to use signing.

  - `Sealing`: Encrypts data using Kerberos. The System.DirectoryServices.AuthenticationTypes.Secure flag must also be set to use sealing.

  - `Delegation`: Enables Active Directory Services Interface (ADSI) to delegate the user's security context, which is necessary for moving objects across domains.

- **ServerBind**: If your ADsPath includes a server name, specify this flag when using the LDAP provider. Do not use this flag for paths that include a domain name or for serverless paths. Specifying a server name without also specifying this flag results in unnecessary network traffic.

- **Login Name**: If applicable, enter the login name required for login to the external LDAP table. A server variable can be used to define the login name as a link to information stored encrypted in the server alias configuration of the Alfabet Server. For information on the definition and specification of server variables, see *Defining Connections Based On Server Variables* in the reference manual *System Administration*.

- **Login Password**: If applicable, enter the login password required for login to the external LDAP table. A server variable can be used to define the login password as a link to information stored encrypted in the server alias configuration of the Alfabet Server. For information on the definition and specification of server variables, see *Defining Connections Based On Server Variables* in the reference manual *System Administration*.

- **Connection String**: Enter the connection string required to connect to the external LDAP table. The connection string depends on the configuration of the external LDAP table.

  ⚠️ It is recommended that you define the database login in the connection string via a user name and password. If Windows Authentication is used for login and the ADIF Console Application is started with a remote alias, the connection to the database will only be successful if the Alfabet Server is started with the same domain user name and access rights that are provided with the connection string.

  ℹ️ Server variables can be used in the connection string. Server variables allow you to define all or part of the definition in the server alias configuration of the Alfabet Server instead of directly defining it in the ADIF scheme. Use of server variables is useful, for example, when using the configuration in a test and production environment with different external sources. The same external source definition can be used in both environments. The server alias definition used in the test and production environment define the correct connection data for the external source used in the respective environment. see *Defining Connections Based On Server Variables* in the reference manual *System Administration*.

- **Is Active** : Select `True` to activate the execution of all import entries within the import set. Select `False` to deactivate the execution.

3) Right-click the LDAP import set 📁 in the explorer and select **Create Entry**. A new import entry is added to the LDAP import set.

4) Right-click the **SQL Commands - DataUpload** folder in the import set and select **Create SQL Command**. A new SQL command ⚡ is added to the folder.

5) Select the new SQL command ⚡ in the explorer and set the following attributes in the attribute window:

- **Name**: Enter a meaningful name for the SQL command.

- **Comments**: Enter a comment that provides information about the functionality implemented by the SQL command.

- **Command Type** : Select `DMLStatement` in the drop-down list.

- **Result Type** : Select `Undefined`.

- **Ignore Errors** : Select `True` if you want the upload to be executed even if the SQL statement specified with the attribute **Text** results in an exception. Select `False` if you do not want the upload to be executed in case of an error in the SQL statement.

- **Is Active** : Select `True` to activate the SQL command. Select `False` to deactivate the SQL command.

- **Text**: Define a LDAP search filter as defined in RFC 2254 that returns a data set when executed on the LDAP table. The resulting data set is uploaded by ADIF as import data. Either write the query directly in the attribute field or click the **Browse** button to open an editor for the definition of the query in the **SQL Text** tab of the editor.

    (i) The text editor for the definition of queries in the ADIF configuration interface provides help to define the query in separate tabs. For more information, see *Defining SQL Queries for SQL Commands* in the section *Configuring ADIF Schemes*.

    (i) If you want to configure complex import scenarios, you can add additional SQL commands to the import entry that are executed prior to or after data upload. For more information about the definition of additional SQL commands within an import entry, see *Configuring SQL Commands for Optional Enhanced Import Features*.

6) Define the import of the data set created via the SQL command for data upload in the import set. For information about the required configuration, see *Defining Import of External Data in an Import Entry*.

7) If more than one data set shall be uploaded from the external database, repeat steps 3.) to 6.).

After creating the LDAP import set, you must now add import entries that map the data in the LDAP table to the data in the Alfabet database. For more information, see *Defining Import of External Data in an Import Entry*.

Optionally, you can define SQL commands that allow you to deactivate this part of the import scheme if specified preconditions have been fulfilled.

For more information see *Configuring the Conditional Execution of Parts of the Import Scheme*.

Defining Data Import From XML Files in an XML Import Set

When importing data from an XML file, the structure of the XML file must be translated to the database table structure. XML files are hierarchically structured and in most cases, more than one table must be created to store the data provided in the XML. Therefore the import from a single XML file requires an import set to be added to the ADIF import scheme as a container for all temporary tables that need to be created and filled with the data from the XML file.

As a general rule, at least one temporary table needs to be created for each XML element type that matches the following conditions:

- The XML element type contains XML attributes

- The XML element type contains child elements without XML attributes and/or text content.

For each temporary table to be defined, an element **Import Entry** must be added to the **XML Import Set**. The import table name is the name of the XML element.

The columns of the temporary table must be mapped to the content of the XML element. The temporary table definition must include one child **Attribute** element for the configuration of a database column. The attribute **Import Column** of the **Attribute** element defines which data from the XML element is written to the column of the temporary table. In the import entry, you must define:

- one **Attribute** element for each XML attribute of the import XML element. The name of the XML attribute must be specified in the **Import Column**

- one **Attribute** element for the text content in the import XML element. <name of the XML element>`_text` must be specified in the **Import Column**

- one **Attribute** element for each child XML element without XML attributes (but with text content) included in the import XML element. The name of the child XML element must be specified in the **Import Column**

> ⚠ When text written to an XML element contains or is completely wrapped into elements, for example included for formatting, the text cannot be read during import. In the following code, for example, the status of the application can not be imported from the text because the status `Retired` is written in a child XML element with its own XML attributes:

```
<Application Name="App1">Active</Application>

<Application Name="App2">Planned</Application>

<Application Name="App3"><Style Format="Bold">Retired</Style></Application>
```

Additionally, columns can be added to the temporary tables that define the hierarchy of the XML elements in the XML file. The import mechanism generates an ID for all XML elements in the hierarchy. To map child XML elements to their respective parents, the following configuration is required:

- In the **Import Entry** defining the temporary table of the parent XML element, an **Attribute** element must be added. The value for **Import Column** must be <element name>_Id.

- In the **Import Entry** defining the temporary table of the child XML element, an **Attribute** element must be added. The value for **Import Column** defined as <parent element name>_Id.

The hierarchy information is usually not imported to object class properties in the target Alfabet database. The hierarchy information is required to be available in the temporary tables in order to identify the parent XML element in SQL commands that are defined for the import of relations between objects.

For more information, see *Import of Reference Arrays with Reference Support Set to True* in the section *Defining Relations*.

For example, data about devices and their assignment to device groups shall be imported from the following XML file:

```
<Groups>

    <DeviceGroup Name="ImportGroup1">

        <Description>Server hosts</Description>

        <Device Name="ImportDevice1" Version="1.0" StartDate="2011.01.01" EndDate="2015.01.05"/>

        <Device Name="ImportDevice2" Version="1.2" StartDate="2011.08.24" EndDate="2014.12.31"/>

    </DeviceGroup>

    <DeviceGroup Name="ImportGroup2">

        <Description>Web relevant devices</Description>

        <Device Name="Backup Server" Version="2.2.1" StartDate="2009.09.30" EndDate="2013.09.01"/>

        <Device Name="ImportDevice1" Version="1.0" StartDate="2011.01.01" EndDate="2015.01.05"/>

    </DeviceGroup>

</Groups>
```

The XML elements in the file require the following configuration in the XML import set:

- **Groups**: No configuration is required. The **Groups** root XML element does not contain any information of its own. This means that no XML attributes or child XML elements without XML attributes will be included.

- **DeviceGroup**: A temporary table must be defined with an **Import Entry** within the ADIF **XML Import Set** .

This XML element stores information in XML attributes and has a child XML element without XML attributes. The XML element also has child XML elements with XML attributes. To import all data about device groups from the file, the temporary table must have one column for the **Name** XML attribute, one for the **Description** XML element and one for the hierarchical ID definition.

- **Device** : A temporary table must be defined with an import entry within the ADIF XML import set .

This XML element stores information in XML attributes. It is assigned to a parent XML element with own XML attributes. To import all data about devices from the file, the temporary table must have one column for each XML attribute and one for the hierarchical ID definition of the parent XML element.

- **Description**: The XML element has no own XML attributes, but text content and a parent XML element with attributes. It is defined as column of the temporary table of its parent XML element.

The resulting import scheme definition is displayed below. The example is a complete configuration of a working import. Therefore, additional configurations for import of the relations are also included:

The tree structure shows:

- DeviceImportFromXML
  - SQL Commands - OnActivate
  - DeviceGroupImport
    - Attributes
      - Name -> NAME -> Name
      - DeviceGroup_Id -> DEVICEGROUP_ID -> None
      - Description -> DESCRIPTION -> Description
    - Relations
    - Indexes
    - SQL Commands - OnActivate
    - SQL Commands - OnStart
    - SQL Commands - DataBind
    - SQL Commands - OnComplete
  - DeviceImport
    - Attributes
      - Name -> NAME -> Name
      - Version -> VERSION -> Version
      - StartDate -> STARTDATE -> StartDate
      - EndDate -> ENDDATE -> EndDate
      - DeviceGroup_Id -> DEVICEGROUP_ID -> None
      - None -> DEVICEGROUP -> None
    - Relations
    - Indexes
    - SQL Commands - OnActivate
    - SQL Commands - OnStart
    - SQL Commands - DataBind
    - SQL Commands - OnComplete

Labels: Import Entry for DeviceGroup · Columns storing hierarchy information · Import Entry for Device · Additional configuration for relation import

ℹ️ It is recommended that you use the semi-automatic interpretation mechanism for import from XML files provided in the ADIF configuration interface instead of performing the configuration steps one-by-one as described in the following. For more information, see *Semi-Automatic Creation of Import Entry Definitions for Import Files*.

To configure data upload from an XML file:

1) In the explorer of the ADIF configuration interface, right-click the ADIF import scheme and select **Create XML Import Set**. A new **XML Import Set** is added to the explorer. A default name is provided for the XML import set. Below the **XML Import Set** , a folder for the creation of SQL commands to be executed on activation of the XML import set is automatically added.

2) In the attribute window of the **XML Import Set** , set the following attributes to define the import file:

   - **Name**: Enter a meaningful name for the XML import set.

   - **Comments**: Enter a comment that provides information about the functionality implemented within the XML import set.

   - **Import XML File**: Enter the name of the XML file from which the data shall be imported. The name must include the file extension.

   - **Source XSD File**: Optionally, enter the name of the XSD file defining the schema for the XML import file. This attribute is information for the ADIF scheme designer only and has no technical impact.

   - **Is Active** : Select `True` to activate the execution of all import entries within the import set. Select `False` to deactivate the execution.

3) Right-click the **XML Import Set** in the explorer and select **Create Entry**. A new **Import Entry** is added to the XML import set.

4) Define the import of an XML element type in your import XML file to a temporary table and the update of a target Alfabet database table with the information in the temporary table.

   For information about the required configuration, see *Defining Import of External Data in an Import Entry*.

5) If more than one XML element matching the criteria for mapping to a temporary table is included in your import XML file, repeat steps 3.) and 4.) for each XML element.

   Optionally, you can define SQL commands that allow you to deactivate this part of the import scheme if specified preconditions have been fulfilled.

   For more information see *Configuring the Conditional Execution of Parts of the Import Scheme*.

Pre-Processing XML files As Part of the Import Process

If the XML structure that you would like to import does not match the required structure for import via an ADIF import scheme, for example because you have formatting elements in the text or elements containing different information in different hierarchy levels of the XML have the same name, you can define an XSL transformation definition in the ADIF import scheme that converts the XML structure of all files to be imported into a format suitable for ADIF import and import the XML structure resulting from the XSL transformation.

If you define the XSL transformation code in the ADIF import scheme prior to defining the ADIF elements of the XML import, the XSL transformation will also be applied to the imported XML prior to executing the functionality **Create XML Set from File**.

To apply XSL pre-processing to imported XML files:

1) In the explorer of the ADIF configuration interface, click the ADIF import scheme for import of your XML. The attribute window opens.

2) Click the **Browse** button in the field of the attribute XSL Transformation. A text editor opens.

3) Define your XSL code in the editor.

> ⚠️ Please note that the term select is not allowed in the XSL file for security reasons. For example, a statement like:
>
> ```
> <xsl:apply-templates select=".//*"/>
> ```
>
> is not allowed in the XSL when assigned to an ADIF import scheme.

4) Click OK to safe your changes.

Defining Data Import from JSON Files

JSON data is imported to a predefined temporary table structure. For each key/value pair within a JSON object, a row is written into the temporary table.

The temporary table has the following columns:

- `OBJID`: An internal ID is assigned to each JSON object in the import file to unambiguously identify the JSON object and to be able to build the hierarchical structure.

- `PARENTOBJID`: If a JSON object is defined within a property field of a parent JSON object, the internal object ID in the column OBJID of the parent object is written into the `PARENTOBJID` column for the subordinate JSON object.

- `PARENTPROPNAME`: If a JSON object is defined within a key of a parent JSON object, this column contains the name of the key.

- `PROPNAME`: The name of the JSON key.

- `PROPISLIST`: A Boolean value that is true if the key contains a list of JSON objects.

- `VALUEINTEGER`: If the value for the key is an integer value, it is written into this column.

- `VALUEFLOAT`: If the value for the key is a float value, it is written into this column.

- `VALUEDATE`: If the value for the key is a date, it is written into this column.

- `VALUETEXT`: If the value for the key is a text or string, it is written into this column.

- `VALUEBOOLEAN`: If the value for the key is a Boolean value, it is written into this column.

The table is filled as follows for different JSON structures. For example:

Import of text, string, integer, float, Boolean and date key/value pairs on the root level

For each key/value pair there is a row in the table.

The `OBJID` for all key/value pairs belonging to the same object is identical.

`PARENTOBJID` and `PARENTPROPNAME` are empty on the root level.

The key is written into the column `PROPNAME` while the value is written into the `VALUE`<data type> column corresponding to its data type.

```
{
    "TextProperty": "This is a root property",
    "DateProperty": "2018-02-13T09:15:00",
    "IntegerProperty": 16
}
```

	OBJID	PARENTOBJID	PARENTPROPNAME	PROPNAME	PROPISLIST	VALUEINTEGER	VALUEFLOAT	VALUEDATE	VALUETEXT
1	1			TextProperty					This is a root property
2	1			DateProperty				13/02/2018 09:15:00	
3	1			IntegerProperty		16			

Import of an array key/value pair on the root level

For each value in the array there is a row in the table.

All values are belonging to the same object and the same key, therefore the object identifier written into the OBJID column and the key name written into the PROPNAME column are identical for all rows.

The column PROPISLIST is set to 1, that means true, because the key value is a list of values. In the following example of the resulting dataset this is converted to an x by the SQL parser.

PARENTOBJID and PARENTPROPNAME are empty on the root level.

The values are written into the VALUE<data type> column corresponding to their data type.

```
{
    "ArrayProperty": ["Value1","Value2"]
}
```

	OBJID	PARENTOBJID	PARENTPROPNAME	PROPNAME	PROPISLIST	VALUEINTEGER	VALUEFLOAT	VALUEDATE	VALUETEXT
1	1			ArrayProperty	x				Value1
2	1			ArrayProperty	x				Value 2

Import of an object list on the root level

For each key/value pair of any of the root objects there is a row in the table.

Each object in the list has a unique OBJID. The OBJID for all key/value pairs belonging to the same object is identical.

PARENTOBJID and PARENTPROPNAME are empty, because all objects are root objects.

The key is written into the column PROPNAME while the value is written into the VALUE<data type> column corresponding to it's data type.

```
[
```

```
    {

        "RootTextProperty": "This is the first object in a list",

        "RootIntegerProperty": 1

    },

    {

        "RootTextProperty": "This is the second object in a list",

        "RootIntegerProperty": 2

    }

]
```

	OBJID	PARENTOBJID	PARENTPROPNAME	PROPNAME	PROPISLIST	VALUEINTEGER	VALUEFLOAT	VALUEDATE	VALUETEXT
1	1			RootTextProperty					This is the first object in a list
2	1			RootIntegerProperty		1			
3	2			RootTextProperty					This is the second object in a list
4	2			RootIntegerProperty		2			

Import of a key/value pair of a root object containing a subordinate object

There is one row for the key containing the object as value and one for each key/value pair of the subordinate object.

In the row for the key containing the object, the object ID of the root object is written into the column `OBJID` and the key is written into the column `PROPNAME`. All other columns are empty.

In the rows for the key/value pairs of the subordinate objects, `PARENTOBJID` and `PARENTPROPNAME` are identical to the `OBJID` and `PROPNAME` column values of the key containing the object.

The `OBJID` of the subordinate object is a unique ID that provides information about the hierarchy in the file. The object ID value starts with the object ID of the root ancestor object and adds the ID of each object in the hierarchy, with the levels separated with a dot. An object on the third level of the hierarchy would have an object ID like 1.1.1.

In the rows for the key/value pairs of the subordinate objects, the key is written into the column `PROPNAME` while the value is written into the `VALUE`<data type> column corresponding to it's data type.

```
    {

        "ObjectProperty":
```

```
{

    "SubTextProperty": "This is a subordinate object",

    "SubIntegerProperty": 1

    }

}
```

	OBJID	PARENTOBJID	PARENTPROPNAME	PROPNAME	PROPISLIST	VALUEINTEGER	VALUEFLOAT	VALUEDATE	VALUETEXT
1	1			ObjectProperty					
2	1.1	1	ObjectProperty	SubTextProperty					This is a subordinate object
3	1.1	1	ObjectProperty	SubIntegerProperty		1			

Import of a key/value pair of a root object containing a list of objects

There is one row for each key/value pair of the subordinate objects.

In each row, the object ID of the object containing the key that returns the list of object is written into the column `OBJID`. The name of that key is written into the column `PROPNAME`.

The `OBJID` of the subordinate objects is a unique ID that provides information about the hierarchy in the file. The object ID value starts with the object ID of the root ancestor object and adds the ID of each object in the hierarchy, with the levels separated with a dot. An object on the third level of the hierarchy would have an object ID like 1.1.1. All rows belonging to the same object have the same ID.

In the rows for the key/value pairs of the subordinate objects, the key is written into the column `PROPNAME` while the value is written into the `VALUE`<data type> column corresponding to it's data type.

```
{

    "ObjectListProperty": [

        {

            "SubTextProperty": "This is the first object in a list",

            "SubIntegerProperty": 1

        },

        {

            "SubTextProperty": "This is the second object in a list",
```

```
            "SubIntegerProperty": 2

        }

    ]

}
```

	OBJID	PARENTOBJID	PARENTPROPNAME	PROPNAME	PROPISLIST	VALUEINTEGER	VALUEFLOAT	VALUEDATE	VALUETEXT
1	1.1	1	ObjectListProperty	SubTextProperty					This is the first object in a list
2	1.1	1	ObjectListProperty	SubIntegerProperty		1			
3	1.2	1	ObjectListProperty	SubTextProperty					This is the second object in a list
4	1.2	1	ObjectListProperty	SubIntegerProperty		2			

To define an import entry for the import of a JSON object:

1) In the explorer of the ADIF configuration interface, right-click the ADIF import scheme and select **Create Entry for Hierarchical JSON**. A new **Import Entry** 🖼 is added to the explorer. A default name is provided for the import entry.

> ℹ The context menu option **Create JSON Import Set** and **Create JSON Set from File** are available for backward compatibility reasons. These options cannot be used for all JSON formats. For example a JSON list of object on the top level of the JSON file cannot be imported via these options. They were therefore substituted with the method **Create Entry for Hierarchical JSON**.

2) In the attribute window of the **Import Entry** 🖼, set the following attributes:

   • **Name**: It is recommended to change the default name for the import entry to a meaningful string that helps you identify the purpose of the import entry.

   • **Import Table**: Enter the name including file extension of the JSON import file from which the data shall be imported. The file extension must be.json.

   • **Temporary Database Table**: Enter a name for the temporary database table to which the data shall be imported. The temporary database table is automatically created in the Alfabet database table during import and filled with the external data. A default is automatically entered when creating the import entry. The name of the temporary table must be unique within the ADIF import scheme.

   • **Comments**: Enter a description of the import process triggered by the JSON import entry.

   • **Is Active** : Select `True` to activate the execution of the import entry. Select `False` to deactivate the execution.

3) The **Attributes** folder of the JSON import entry automatically contains the required attribute entries to create the predefined temporary table structure for JSON import. The attributes must not be changed.

4) For each database table that shall be updated with data in the JSON import, create an **Import Entry** 🗔 as defined in the section *Defining Import of External Data in an Import Entry*. The data defined in the temporary table of the JSON import entry can be used as input for the temporary tables in other import entries as follows:

- Leave the import column of each attribute defined for the import entry empty.

- Create an SQL command of the type **OnStart** defining an UPDATE or INSERT INTO statement to fill the temporary table of the current entry with the data from the temporary table defined in the JSON import entry.

ⓘ   For details about the required configuration, see *Defining Import of External Data in an Import Entry* and the example below.

◉   The following example describes an import of application group short names via a JSON file with the following structure:

```
[
    {
    "ClassName": "ApplicationGroup",
    "Values": {
        "name": "OptiRetail CRM",
        "shortname": "OR CRM"
        }
    },
    {
    "ClassName": "ApplicationGroup",
    "Values": {
        "name": "AI CRM AS-IS Situation",
        "shortname": "AI CRM IS"
        }
    }
```

```
        ]
```

Both application groups listed in the example JSON do not have a short name in the Alfabet database:

	REFSTR	NAME	SHORTNAME
1	95-2-0	OptiRetail CRM	
2	95-21-0	AI CRM AS-IS Situation	

The ADIF import scheme contains two entries:

- `JSONImport` for import of the JSON file to the predefined temporary table.

- `AppgImport` for update of the `APPLICATIONGROUP` table of the Alfabet database.

The import entry `AppgImport` defines the temporary table and the mapping to the object class properties of the class application group via the settings of the import entry and the definition of attributes in the **Attributes** folder. The attributes are displayed as:

column in import file -> column in temporary table -> column in Alfabet object class table

The import column in the attributes is undefined, because import is not performed via an external data source. The temporary table will be filled via SQL commands of the type `OnStart`.

The temporary table has three columns:

- `IDVAL`, which will be used to map the data to one of the JSON objects in the JSON file. This column is not imported into the Alfabet object class table.

- `APPGNAME`, which is defined as class key and mapped to the object class property `Name` of the class `ApplicationGroup`. The ADIF import mechanism will use the data in this column to map the data in the temporary table to the correct application group.

- `APPGSHORTNAME`, which is mapped to the object class property `ShortName` of the class `ApplicationGroup`.

In the **SQL Commands - OnStart** folder, two SQL commands are defined that will be executed after the import of the data from the import file and prior to the data bind that maps the temporary tables of the import entries to Alfabet objects, if applicable.

ExampleJSONImport
- Parameters
- JSONImport
  - Attributes
    - None -> OBJID -> None
    - None -> PARENTOBJID -> None
    - None -> PARENTPROPNAME -> None
    - None -> PROPNAME -> None
    - None -> PROPISLIST -> None
    - None -> VALUEINTEGER -> None
    - None -> VALUEFLOAT -> None
    - None -> VALUEDATE -> None
    - None -> VALUETEXT -> None
    - None -> VALUEBOOLEAN -> None
  - Relations
  - Indexes
  - Indicators
  - Roles
  - Events
  - SQL Commands - OnActivate
  - SQL Commands - OnStart
  - SQL Commands - DataBind
  - SQL Commands - OnComplete
- AppgImport
  - Attributes
    - None -> APPGNAME -> Name
    - None -> APPGSHORTNAME -> ShortName
    - None -> IDVAL -> None
  - Relations
  - Indexes
  - Indicators
  - Roles
  - Events
  - SQL Commands - OnActivate
  - SQL Commands - OnStart
    - CMD_updatenameid
    - CMD_updateshortname
  - SQL Commands - DataBind
  - SQL Commands - OnComplete

Import entry for import from JSON to a temporary table

The import entry settings define the mapping to the object class and the name of the temporary table.

The attribute settings define the column name of the temporary table as well as the mapping to an object class property of the object class ApplicationGroup . No import column is defined.

SQL commands to write data from the temporary table created for the JSONImport entry into the temporary table of the AggfImport entry

Class	
Class Name	ApplicationGroup
Create Instances	True
Update Instances	True
Use Update Properties	False
Common	
(Name)	AppgImport
Assistant	
Events	(Collection)
User Data	
Data	
Import Column Binding	ByName
Import Start Row	0
Import Table	
Read Parameter	True
Squeeze Audit Table	False
Table Owner	True
Temporary Database Table	APPGROUPIMPORT_TMP

Common	
Back Relation	
Data Type	String
Database Column	APPGNAME
Import Column	
Is Class Key	True
Language	
Property Name	Name
Rescan Back Relations	True
Size	255
To Class	
Trim End	True
Trim Start	False
Truncate to Size	False

During import, first all temporary tables are created and then import from the example JSON file is performed for the `JSONImport` JSON import entry. This leads to the following dataset in the temporary table `TMP_JSON_IMPORT`:

	OBJID	PARENTOBJID	PARENTPROPNAME	PROPNAME	PROPI...	VA...	V...	VA...	VALUETEXT
1	1			ClassName					ApplicationGroup
2	1			Values					
3	1.1	1	Values	name					OptiRetail CRM
4	1.1	1	Values	shortname					OR CRM
5	2			ClassName					ApplicationGroup
6	2			Values					
7	2.2	2	Values	name					AI CRM AS-IS Situation
8	2.2	2	Values	shortname					AI CRM IS

Afterwards, the import mechanism executes the SQL commands in the `AppgImport` import entry in the given order. The first SQL command searches the `TMP_JSON_IMPORT` table for rows where the `PROPNAME` is `name` and writes values for `VALUETEXT` and `OBJID` of these rows into the temporary table `AP-PGROUPIMPORT_TMP`:

```
INSERT INTO APPGROUPIMPORT_TMP (APPGNAME, APPGSHORTNAME, IDVAL)

SELECT VALUETEXT, NULL, OBJID

FROM TMP_JSON_SRC

WHERE TMP_JSON_SRC.PROPNAME ='name'
```

	APPGNAME	APPGSHORTNAME	IDVAL	REFSTR
1	OptiRetail CRM		1.1	
2	AI CRM AS-IS Situation		2.2	

The column `REFSTR` that is visible in the temporary table has been added by the ADIF import mechanism to perform the mapping to application groups in a later step.

The second SQL command searches the `TMP_JSON_IMPORT` table for rows where the `PROPNAME` is `shortname` and updates the existing rows in the `AP-PGROUPIMPORT_TMP` table by writing the values in the `VALUETEXT` column into the column `APPGSHORTNAME`, using the `OBJID` and `IDVAL` columns in the two tables to match the data to the correct row.

```
UPDATE APPGROUPIMPORT_TMP SET APPGROUPIMPORT_TMP.APPGSHORTNAME = TMP_JSON_SRC.VALUETEXT FROM APPGROUPIMPORT_TMP,
TMP_JSON_SRC WHERE TMP_JSON_SRC.OBJID = APPGROUPIMPORT_TMP.IDVAL AND TMP_JSON_SRC.PROPNAME ='shortname'
```

	APPGNAME	APPGSHORTNAME	IDVAL	REFSTR
1	OptiRetail CRM	OR CRM	1.1	
2	AI CRM AS-IS Situation	AI CRM IS	2.2	

The ADIF import mechanism now maps the entries in the table with existing application groups and fills the REFSTR of the application groups into the REF-STR column of the APPGROUPIMORT_TMP table:

	APPGNAME	APPGSHORTNAME	IDVAL	REFSTR
1	OptiRetail CRM	OR CRM	1.1	95-2-0
2	AI CRM AS-IS Situation	AI CRM IS	2.2	95-21-0

In the next step, the ADIF import mechanism uses the information in the REFSTR and the APPGSHORTNAME columns of the APPGGROUPIMPORT_TMP table to update the standard Alfabet database table APPLICATIONGROUP with the correct short name per object:

	REFSTR	NAME	SHORTNAME
1	95-2-0	OptiRetail CRM	OR CRM
2	95-21-0	AI CRM AS-IS Situation	AI CRM IS

After that, the temporary tables are removed from the database.

Defining Data Import from Microsoft® Excel Files and Comma-Separated File Formats

When data is imported from CSV. TXT, XLSX or XLS files, the data is written from the import file directly to the temporary table in the Alfabet database. The data in the import file should correspond to a database table structure:

- **Interpretation of Comma-Separated File Formats**

The CSV file is regarded as a representation of a database table with the database column headers specified in the first line of the file. The following lines in the file represent the data rows in the database table.

- **Interpretation of Excel® File Formats**

Data is read from the first sheet in the file only. The data in the Excel® must have a flat table structure. The column headers must be specified in the first row. The following rows in the table represent the data rows in the database table.

When importing from Microsoft® Excel® files or comma-separated file formats, the import entries that define how data is processed during import can be defined directly in the import scheme. Nevertheless, it is possible to define grouping within file import sets. The file import set is like a folder node in the ADIF explorer. The only functionality provided by the folder is the batch activation or deactivation of all import entries within the file import set.

To create a file import set:

1) In the explorer of the ADIF configuration interface, right-click the ADIF import scheme and select **Create File Import Set**. The new **File Import Set** is added to the explorer with a default name for the file import set. Below the **File Import Set** , a folder for the creation of SQL commands to be executed on activation of the import set is automatically added.

2) In the attribute window of the file import set , set the following attributes to configure the connection to the external database:

   • **Name**: Enter a meaningful name for the file import set.

   • **Comments**: Enter a comment that provides information about the functionality implemented within the file import set.

   • **Is Active** : Select `True` to activate the execution of all import entries within the import set. Select `False` to deactivate the execution.

After creating the file import set, you must now add import entries that map the data in external files to the data in the Alfabet database.

For more information, see *Defining Import of External Data in an Import Entry*.

Optionally, you can define SQL commands that allow you to deactivate this part of the import scheme if specified preconditions have been fulfilled. For more information see *Configuring the Conditional Execution of Parts of the Import Scheme*.

Defining Import of External Data in an Import Entry

An import entry defines the import of data to the Alfabet database for one single object class from one single source. You can define multiple import entries for a single import file or database table in order to import multiple classes from one source. Even if the files all have the same file type, you must define multiple import entries to import data for one class from multiple database tables or files.

When importing data from Microsoft® Excel® files or comma-separated file formats, each import file is regarded as a database table. Therefore, the data in the file must correspond to the structure of a database table.

- **Interpretation of Comma-Separated File Formats**

The CSV file is regarded as a representation of a database table with the database column headers specified in the first line of the file. The following lines in the file represent the data rows in the database table.

- **Interpretation of Excel® File Formats**

Data is read from the first sheet in the file only. The data in the Excel® must have a flat table structure. The column headers must be specified in the first row. The following rows in the table represent the data rows in the database table.

- **Interpretation of External Database Tables**

The data set that is used as import data is not directly read from the database tables of the external database. In the ADIF import entry defining the import, an SQL query with a SELECT statement must be defined as an SQL command for data upload. The data set resulting from the query is uploaded by ADIF as import data. The column headers of the import table are defined by the alias specifications in the SELECT statement.

For more information about data upload from an external database and the configuration required before carrying out the configuration described below, see *Defining Data Upload from an External Database in a Database Import Set*.

- **Interpretation of LDAP Tables**

The data set that is used as import data is not directly read from the LDAP tables database. In the ADIF import entry defining the import, a query must be written in LDAP-specific query syntax and must be defined as an SQL command for data upload. The data set resulting from the query is uploaded by ADIF as import data. The column headers of the import table are defined by the data set header definitions in the query. For more information about data upload from LDAP and the configuration required prior to performing the configuration described in this section, see *Defining Data Upload from an LDAP Table in an LDAP Import Set*.

- **Interpretation of XML files**

Each XML element that contains XML attributes and/or child XML elements with text content but without own XML attributes is regarded as a database table. The database table name is the name of the XML element. The database column names are the names of the XML attributes of the XML element as well as the names of the child elements without own XML attributes but with text context.

To store information about the hierarchy of the XML elements available in the import file in the temporary tables generated during ADIF import, an ID is automatically created for each XML element during import. A database column storing the ID of the XML element must be added to each temporary table and a database

column storing the ID of the parent XML element must be added to the temporary table for each XML element that has a parent XML element.

An import entry triggers the import by means of two steps:

1) External data is written to a temporary table

2) Data from the temporary table is written to the Alfabet database table.

The information required to build the temporary table and the changes to be performed in the Alfabet database table must be specified in the child elements of the **Import Entry** . The temporary table does not necessarily have to be an exact copy of the data as provided in the import database or file. You can add additional columns to the table that are left empty or filled with either a static value or a value that is specified by means of an SQL query. Data can be selected from the initial temporary database tables in order to create new temporary tables.

As a final step, you configure which temporary tables are used to change the data in the Alfabet database table of the target object class.

Configuring the import to the Alfabet database can be a complex process that includes definition of various child elements of the import entry and SQL commands to be executed during the process. The following sections provide basic information about how to configure an import entry and its child elements in the context of basic tasks that are most commonly part of an import process. When defining import entries, you must combine and adapt the different possible configuration steps to match your individual requirements.

It is important to take the order of execution of the import entries in the ADIF import scheme into account when configuring import. The import process reads all import entries multiple times to perform the following consecutive actions:

1) Create all temporary tables defined in all import entries in the order of the **Import Entry** definitions in the ADIF import scheme.

2) Write data sets from the external files/database tables to the temporary tables in the order of import entry definition in the ADIF import scheme.

3) Perform all DML statements defined in SQL commands of the type `OnStart`.

4) Update the Alfabet object class database tables with data from the temporary tables according to the specification in the import entries in the order of import entry definition in the ADIF import scheme.

5) Update the RELATIONS table of the Alfabet database if relation updates are defined in import entries in the order of import entry definition in the ADIF import scheme.

6) Perform all DML statements defined in SQL commands of the type `OnComplete`.

7) Drop all temporary tables if that the ADIF import scheme is configured to drop temporary tables.

ⓘ The ADIF import scheme allows the specification of additional SQL commands to be executed at different times during the import process. For information on the order of execution of these SQL commands, see the section *Configuring SQL Commands for Optional Enhanced Import Features*.

This order of execution allows, for example, data to be found and processed from all temporary tables created for other import entries in the SQL commands of an import entry.

Mapping External Data to Temporary Tables and Alfabet Database Tables

Prior to configuring your import, you must first evaluate how the external data is mapped to the Alfabet meta-model and the Alfabet database tables

- *What is the target class for my data import?*

One import file can include data for import to multiple classes. You must then define multiple import entries for import from the same data. Each import entry defines a temporary table that imports the subset of data relevant for import to one object class of the Alfabet meta-model.

- *Does the target class have the required properties to store the data to be imported for the class?*

If the import data includes a property that is not available in the standard object class properties of the target object class in the Alfabet meta-model, new properties can be created in the Alfabet meta-model for the target object class.

> (i) Custom properties must be configured in the configuration tool in the configuration tool Alfabet Expand. For information about how to create a custom property, see the section *Configuring Custom Properties for Protected or Public Object Classes* in the reference manual *Configuring Alfabet with Alfabet Expand*.

> (!) You must use the configuration capabilities in Alfabet Expand to add new property columns to a database table of a Alfabet object class. It is not permissible to add a property column via a DDL statement in the SQL commands of the ADIF scheme. **Existing standard Alfabet database tables must not be altered by SQL commands via ADIF.**

- *What are the required data formats and types for the import data?*

Check whether the import data matches the data format of the target database table column in the Alfabet database. the data types and formats are described in the table Data Types and Formats in the chapter The Alfabet Meta-Model in the Alfabet Database.

The ADIF import scheme provides mechanisms to automatically adapt date, number, and boolean values to the formatting required in the Alfabet database. You can provide information about the data format used in your external data for dates, numbers and Boolean expressions and these values are then automatically converted to the right format during import.

For more information, see *Defining Import of External Data in an Import Entry*.

The ADIF import scheme also allows empty columns to be added to the temporary table and fills them with values via SQL commands after the external data has been written to the temporary table. You can also define SQL commands to alter data that has been written to the temporary database table. These mechanisms allow you to convert import data to any desired import format or to set default values if the import data does not consistently provide information for a property.

For more information, see *Use Case: Filling Empty Property Values with Default Values When Importing to Mandatory Object Class Properties*.

> (i) Empty strings cannot be imported to the Alfabet database. If an empty string is imported for a property value, the property value is set to NULL in the Alfabet database. If you do not want NULL to be set for the property, you can use SQL commands to set default values instead of empty strings in the temporary tables before import to Alfabet database tables.

- *When new objects are created during import, does the import data provide all mandatory information?*

Some object class properties in the Alfabet meta-model are mandatory. When such a property is missing during import, you may corrupt your Alfabet database. Please note that mandatory properties are highlighted yellow in the meta-model browser in the ADIF configuration interface. You must ensure that data is provided for all mandatory properties.

Apart from the object class properties marked as mandatory, other properties might be mandatory because of your specific demands or shall be set in order to allow users to find the imported object. For example, you can import an object without specification of an authorized user, assignment, or any other definition that is used to constitute access permissions for users. As a result, none of the users will have edit rights to the imported object.

If values are not provided for mandatory properties for some objects, you can configure the import entry via SQL commands to set property values in the temporary table prior to data import. For more information, see *Use Case: Filling Empty Property Values with Default Values When Importing to Mandatory Object Class Properties*.

- *Is data translation activated for the target Alfabet database?*

The data translation functionality of the Alfabet database allows to translate object names and descriptions. If the feature is activated, two new columns called `NAME_<language code>` and `DESCRIPTION_<language code>` are added to the database tables for each language that data translation is activated for. The translated values must be imported via ADIF together with the original values. In the ADIF import scheme mapping of data to the properties of the target object class includes a language specification. For each imported language a data mapping must be defined.

> For example, if the Name property of a target class is translated to German, the original and translated values must both be mapped to the Name property of the target class. The original values are mapped without a language specification. The German values are mapped with a German language specification.

For more information, see *Mapping External Data to Temporary Tables and Alfabet Database Tables*.

> For more information about the configuration of object data translation, see the section *Configuring the Translation of Object Data* in the chapter *Localization and Multi-Language Support for the Alfabet Interface*.

- *How can the data records be mapped to existing objects in the target database table?*

The Alfabet meta-model stores data about each object in the Alfabet database with a number of unique identifiers that are automatically set when a new object is created. During data import, these unique identifiers are automatically added to new records with the exception of the unique `REFSTR` property. This property is used to map the imported data records to data records that already exist in the Alfabet database. Therefore, the `REFSTR` of the objects is set during import via a special data bind mechanism that takes mapping to existing data into account. The `REFSTR` of objects is exclusively set by this mechanism. Therefore, data bind must also be performed for initial data import to empty database tables.

The data bind mechanism requires that each object in the external data has a unique identifier that is included in the import. When data is imported to database tables that already contain objects, this identifier must be a unique property of the target object class that allows to identify whether the imported object is identical to an existing object.

The following section describes the mechanisms provided via ADIF for the secure import of data. Additionally, you can use your own SQL commands to add complex functionality to the basic import.

For more information about additional SQL commands, see *Configuring SQL Commands for Optional Enhanced Import Features*.

⚠️ The ADIF import mechanisms are designed to provide a high level of security. Always check whether an import can be performed via an import mechanism provided by ADIF prior to defining your own SQL commands to perform the import.

Creating an Import Entry to Map the Database Tables

You must add at least one import entry per external file or external database table to the ADIF import scheme. An import entry defines mapping between one external file or external database table and one target database table of the Alfabet meta-model.

1) In the explorer of the ADIF configuration interface, right-click the ADIF import scheme or the import set that you want to create an import entry for and select **Create Entry**.

2) In the attribute window of the **Import Entry** , set the following attributes to define the import entry:

   - **Name**: It is recommended to change the default name for the import entry to a meaningful string that helps you identify the purpose of the import entry.

   - **Comments**: Enter a description of the import process configured by the database import set.

   - **Is Active** : Select `True` to activate the execution of the import entry. Select `False` to deactivate the execution.

   - **Squeeze Audit Table**: Select `True` to clean the audit table from meaningless entries created for technical reasons during import.

For more information about the squeezing of audit tables, see *Clean-up of the audit tables*.

3) Define the import to the temporary database table in the attribute window:

   - **Import Table**: Set this attribute according to the requirements of your import format:

     - For import from an XLSX, XLS, JSON, CSV or TXT file: Enter the name including the file extension of the file from which the import shall be performed.

     - For import from an XML file: Enter the name of the XML element in the XML import file for which the data shall be imported for.

     - For import form an external database or LDAP: This attribute will be ignored.

   - **Temp Database Table**: Enter a name for the temporary database table to which the data shall be imported. The temporary database table is automatically created in the Alfabet database table during import and filled with the external data.

     ⚠️ Please note the following:

       - The database table name must be unique. **You must not use the name of an existing Alfabet database table!** It is recommended that you start all

temporary table names with the prefix TMP_ to make sure that no conflicts with the existing database configuration occur.

- For import from JSON, the table name is preconfigured and has to be TMP_JSON_SRC.

- **Table Owner**: Select `True` to activate the automatic creation of the temporary table and the execution of data record mapping with data in the Alfabet database.

> (i) Setting this attribute to `False` in order to deactivate the processing of temporary tables is only relevant when configuring the data import from multiple external files/database tables to a single temporary table.
>
> For more information, see *Collecting Data from Multiple External Files or Database Tables in one Temporary Table*.

4) Define the data formats in the import data to enable automatic conversion to the formats required in the Alfabet database:

- **Boolean'False'** : Enter the value in your external data that shall be interpreted as the Boolean value `False` during import. For example, if you define "no", and the ADIF scheme is configured to import a property as boolean, the value "no" will be automatically converted to the Boolean format `False` used in the Alfabet meta-model during import.

- **Boolean'True'** : Enter the value in your external data that shall be interpreted as he Boolean value `True` during import. For example, if you define "yes", and the ADIF scheme is configured to import a property as boolean, the value "yes" will be automatically converted to the Boolean forma `True` used in the Alfabet meta-model during import.

- **Date Format**: Enter the date format used in your external data.

- **Number Decimal Digits**: Enter the number of decimal digits used in your import data.

- **Number Decimal Separator**: Enter the decimal separator used in your import data in numbers.

- **Number Group Separator**: Enter the group separator used in your import data in numbers.

- **Encoding**: For import from comma-separated files only: In the drop-down list, select the encoding of the import file.

- **Delimiter**: For import from comma-separated files only: Define the delimiter used in the file to separate data within a record.

5) In the attribute window, define the mapping of data records to the target Alfabet database table:

> (i) The following specifications are optional. You can define an import entry for import to a temporary table only (for example, if you want the data to be used exclusively in SQL commands of other import entries). When data from the temporary database table is imported to a target Alfabet database table, these specifications are required.

- **Class Name**: Select the target object class of the Alfabet meta-model.

- **Create Instances** : Select `True` to allow new records to be added to the Alfabet database tables during import triggered by this import entry. Select `False` to limit import to manipulation of existing records.

(i) This specification only applies to the database table of the object class specified with the **Class Name** attribute. If objects are added to other database tables via SQL commands, for example to create indicators or roles assigned to the target object class during import, objects of the related object classes will be created even if the **Create Instances** attribute is set to `False`.

- **Update Instances**: Select `True` to allow existing data to be altered in the Alfabet database tables during import by this import entry.

(i) This specification only applies to the database table of the object class specified with the **Class Name** attribute. If objects are altered in other database tables via SQL commands, for example to update indicators or roles assigned to the target object class during import, objects of the related object classes will be updated even if the **Update Instances** attribute is set to `False`.

Creating Attribute Entries to Map the Database Table Columns

The columns of the temporary table and the mapping of the temporary table columns to both import data and to target columns in the Alfabet database are defined in the child elements attribute of the **Import Entry** . For each column of the temporary table, an attribute element must be defined:

⚠ The specification of database column names is case-sensitive.

1) In the explorer of the ADIF configuration interface, right-click the **Import Entry** that you want to create an **Attribute** element for and select **Create Attribute**.

2) In the attribute window of the **Attribute** element , set the following to define a column in the temporary table:

- **Database Column**: Enter a name for the database column of the temporary table.

- **Import Column**: Enter the name of the column of the external data table from which data shall be written. Setting this attribute is optional. You can define columns in the temporary table that are not filled from the external data.

- **Property Name**: In the drop-down list, select the property of the target object class for which you want to update data. The **Name** attribute of the object class property from the Alfabet meta-model must be used to specify the object class. This attribute is not necessarily identical to the name of the database table column. Setting this attribute is optional. You can define columns in the temporary table that are not mapped to a Alfabet database table column.

⚠ The specification of the target object class property via the **Name** attribute of the object class property limits data import to columns in the target Alfabet database table that contain property definitions. If you want to update data in database table columns that are created by database triggers in the Alfabet database (for example, the database columns for the translation of object names and descriptions to defined languages), you must define the update via an SQL command in the folder **SQL Commands - OnComplete**.

For more information, see *Configuring SQL Commands for Optional Enhanced Import Features*.

- **Is Key**: Select `True` if this property shall be used to map data records to existing data records in the Alfabet database during import. Automatic mapping of data is only performed if at least one attribute element of the import entry has a **Is Key** attribute set to `True`.

- For details about the mapping of objects, see *Defining Mapping of Data Records*.

3) During import, data must be written to the temporary table in a format that matches the requirements of the target database table column of the Alfabet database. Most of the following attributes are set automatically to the required values when you set the **Property Name** attribute. You can optionally adapt them to your specific requirements:

- **Data Type**: In the drop-down list, select the data type of the target object class property in the Alfabet database.

- **Size**: If `String` is selected in the **Data Type** field, define the size of the string column in the temporary database table. The size specification should match the size specification of the target object class property in the Alfabet database.

  > (i) The **Size** attribute can only be set manually if the column of the temporary table is not mapped to a Alfabet database column (by setting the attribute **Property Name**). If **Property Name** is specified, the **Size** value is automatically set to the required size and is reset to the defined value when changed manually.
  >
  > For technical reasons, the **Size** value must be set to <size in the target database table>+1. For example, if you want to import a value with a permissible size of 16, you must define the **Size** attribute as 17.

- **Truncate To Size**: For properties of the **Data Type** `String` only. Select `True` to truncate imported strings with a size greater than the **Size** specification. Select `False` to cancel import with the exception of when import data does not match the specified size.

- **Trim End**: If this attribute is set to `True`, blank spaces at the end of imported strings are removed when importing the data into the temporary table. Please note that non breaking spaces are not removed by this mechanism. By default, the attribute is set to `False`.

- **Trim Start**: If this attribute is set to `True`, blank spaces at the beginning of imported strings are removed when importing the data into the temporary table. Please note that non breaking spaces are not removed by this mechanism. By default, the attribute is set to `False`.

4) If the target property establishes a reference to another object class and the Alfabet meta-model includes a back-reference from the referenced object class to the referencing object class, the back reference is updated automatically during import when the following attributes are defined:

- **Rescan Back Relations** : Select `True` to automatically set back relations.

  > ⚠ Always set **Rescan Back Relations** to `True` when importing to an object class property of the type `Reference` for which the **BackReference** attribute defines an attribute name. Otherwise, your import will result in database inconsistencies.

- **To Class**: In the drop-down list, select the target object class in the Alfabet database that the back reference is to be set for. The **Name** attribute of the object class from the Alfabet meta-model must be used to specify the object class. This attribute is not necessarily identical to the name of the database table. Setting this attribute is optional. If the target class of the

reference is defined in the **TypeInfo** attribute of the property mapped with this attribute element, it is not required to set this attribute.

- **Back Relation**: Enter the name of the target object class property that establishes the back reference. Setting of this attribute is optional. If the property establishing the back reference is already specified in the **BackReference** attribute of the property mapped with this attribute element, it is not required to set this attribute.

5) If the imported data is a translated value for a Name or Description property of the target object class, the target language must be defined with the following attribute:

- **Language**: Select the culture for that the values are imported in the drop-down list.

> ⓘ The **Language** attribute shall not be set for values imported for the standard original language culture.
>
> For more information about the storage of object property translations in the Alfabet database, see Storage of Translatable Object Class Properties in the chapter The Alfabet Meta-Model in the Alfabet Database.

Defining Mapping of Data Records

The unique `REFSTR` property of objects in the Alfabet database is used to map objects to existing objects in the Alfabet database.

The ADIF import scheme provides a data bind mechanism that adds the `REFSTR` property to external data in the temporary database tables, taking mapping of objects to existing data records in the Alfabet database into account. For existing objects, the `REFSTR` configured for the object in the Alfabet database table is written to the temporary table and for new objects, a unique `REFSTR` is generated and added to the temporary table. Based on the `REFSTR` added to the temporary table, import to Alfabet database tables and definition of relations between objects can be performed.

> ⚠ The data bind mechanism must be run during import, even if the database table to that data is imported is empty. While all other unique properties, like for example the `INSTGUID` of new objects is set automatically when writing the new record to the database table, the `REFSTR` property in the Alfabet database tables is exclusively set by the data bind mechanism.
>
> Correct update and creation of objects in the object class database tables of the Alfabet database, including setting of required GUIDs and specification of the last update and creation time and user information depends on the data bind. ADIF import must be handled via the temporary table. Using ADIF for direct import of data into object class database tables of the Alfabet database via SQL commands can lead to a loss of database integrity and must not be performed.

> ⓘ All properties in the group `Artifact` have properties that store information about changes to the object.
>
> The following properties are set when an object is created or updated via the data bind based ADIF import:
>
> - The property `LAST_UPDATE` is set to the current date.
>
> - The property `LAST_UPDATE_USER` is set to the user triggering the update. If a user cannot be identified, the property is set to `ALFABET_INTERN`.

The following properties are additionally set when an object is created:

- The property CREATION_DATE is set to the current date.

- The property CREATION_USER is set to the user triggering the update. If a user cannot be identified, the property is set to ALFABET_INTERN.

If an audit table is available for an object class, the audit history is also updated. For more information, see Audit History Storage.

The data bind mechanism does the following:

- A column REFSTR is automatically added to the temporary database table.

- After import of the external data to the temporary table, the data bind mechanism compares the data records in the temporary database table with data records in the target Alfabet database table by means of customer defined key properties. If records in the temporary table are identified as matching a record in the Alfabet database table, the REFSTR from the Alfabet data record is written into the REFSTR column of the temporary database table.

- For each row in the temporary table that a REFSTR is defined for, the data from the temporary table is used to update the data with a respective REFSTR in the Alfabet database table.

- New objects are added to the target Alfabet database table for all rows in the temporary database table that do not have a REFSTR defined. During creation of the new objects, the REFSTR property is automatically set in the target Alfabet database tables.

- The REFSTR of the new objects is added to the temporary database table. When processing of the bind command of the import entry is finished, the REFSTR column of the temporary table is completely set with the correct REFSTR of all objects.

Data bind requires the following definition:

- The name of the target object class in the Alfabet meta-model must be defined in the attribute **Class Name** of the import entry.

- At least one element **Attribute** with the attribute **Property Name** defining the **Name** of a property of the target object class must be defined for the import entry.

- The mapping conditions must be defined. There are two ways to define mapping conditions:

  - The attribute **Is Key** is set to True for at least one element **Attribute** with a **Property Name** definition. The ADIF data bind mechanism then executes an automatically created bind query that maps the content of the key column in the temporary database table with the content of the column in the Alfabet database table specified with the attribute **Property Name**. You can set the attribute **Is Key** to True for multiple elements **Attribute**. The bind mechanism then maps records if all defined conditions apply.

  - The specification of the data mapping is done via a customer defined SQL query that is defined in the folder **SQL Commands - DataBind**.

Both mechanisms can be used in parallel.

It is recommended to rely on the mapping via key attributes whenever possible.

To define an own SQL command for data bind:

1) In the explorer of the ADIF configuration interface, navigate to the relevant **Import Entry** .

2) Right-click the folder **SQL Commands - DataBind** and select **Create SQL Command**. A new SQL command ⚡ is added as child node of the folder.

3) Click the new SQL command ⚡ and specify the following attributes in the attribute window:

- **Name**: Enter a meaningful name for the SQL command.

- **Comments**: Enter a comment that provides information about the functionality implemented by the SQL command.

- **Command Type**: Select `DMLStatement`.

- **Result Type** : Select `Undefined`.

- **Ignore Errors** : Select `True` if you want the export to be executed even if the SQL statement specified with the attribute **Text** results in an exception. Select `False` if you do not want the export to be executed in case of an error in the SQL statement.

- **Is Active** : Select `True` to activate the SQL command. Select `False` to deactivate the SQL command.

4) In the attribute window, click the **Browse** ⊞ button in the **Text** attribute to open an editor for the definition of the SQL query. Define the SQL query in the **SQL Text** tab of the editor.

> ⓘ The text editor for the definition of SQL queries in the ADIF configuration interface provides help for the definition of the SQL query in separate tabs. For more information, see *Defining SQL Queries for SQL Commands* in the chapter *Configuring ADIF Schemes*.

The SQL query must be an `UPDATE` statement targeting the temporary database table. The update statement must be defined to fill the `REFSTR` column of the temporary database table with the `REFSTR` value of matching objects in the target database table of the Alfabet database. The import of new objects for which no matching object has been found in the target Alfabet database table and the setting of the `REFSTR` for new objects is done automatically by the bind mechanism after executing the bind query.

> 👁 For example, a typical bind SQL query for the import of device groups mapped to existing objects with the same name:
>
> ```
> UPDATE TMP_DEVICEGROUP
>
> SET REFSTR = (SELECT dg.REFSTR FROM DEVICEGROUP dg WHERE
> dg.NAME = TMP_DEVICEGROUP.NAME)
> ```

Defining Relations

Relations between objects of different classes are stored in object class properties of the data type `Reference` or `ReferenceArray` in the Alfabet database. Setting of references between objects in the Alfabet database s can be included in the ADIF import.

> ⚠ When importing references between objects, you must ensure that an object does not reference itself. Loop references can cause severe problems in the Alfabet database.

If you want to set a reference between objects, you must first evaluate how the relation is established in the Alfabet database:

- *Is the relation part of the standard Alfabet meta-model?*

If neither the referencing nor the referenced object class contains an appropriate object class property of the type Reference or `ReferenceArray`, you must define a custom property of the type Reference or `ReferenceArray` to establish the relation.

> ⓘ Custom properties must be configured in the configuration tool Alfabet Expand. For information about how to create a custom property, see the section *Configuring Custom Properties for Protected or Public Object Classes* in the reference manual *Configuring Alfabet with Alfabet Expand*.

> ⚠ You must use the configuration capabilities in Alfabet Expand to add new property columns to a database table of a Alfabet object class. It is not permissible to add a property column via a DDL statement in the SQL commands of the ADIF scheme. **Existing standard Alfabet database tables must not be altered by SQL commands via ADIF.**

- *What is the data type of the property establishing the relation?*

Relations of the type Reference establish a relation with one object only and are configured in an **Attribute** 🔲 element in the import entry for the referencing class. Relations of the type `ReferenceArray` establish a 1 to n relation to multiple objects. If the attribute **Reference Support** is set to `False`, the relation is configured in an **Attribute** element in the import entry for the referencing class. For more information, see *Defining Relations*.

If the attribute **Reference Support** is set to `True`, the relation is configured in a **Relations** element in the import entry of either the referenced or referencing object. For more information, see *Import of ReferenceArrays with Reference Support Set to True*.

- *Is a back relation available?*

For a small number of properties of the type Reference, a database trigger in the Alfabet database automatically sets a corresponding back reference in a property of the referenced object class. If a property leads to a creation of back references, the attribute **Back Reference** of the object class property displays the name of the property of the referenced class that is updated by the trigger. If a back reference is available, **Rescan Back Relations** must be set to `True` in the ADIF **Attribute** element defining the relation to automatically update the back relation.

- *Are all objects that are target of the relation available in the Alfabet database ?*

Relations can only be established to existing objects in the Alfabet database. If target objects are missing, they must be created in the Alfabet database prior to import. You can, for example, configure the ADIF import scheme to create the missing target objects in the respective Alfabet database table. Make sure that the creation of objects takes place prior to the setting of relations during the import process.

Establishing relations between objects requires different configuration in ADIF depending on the reference type.

Special support is provided for the references that must be imported to define an indicator or a role for an object. The definition of a role or an indicator for an object requires the definition of multiple references between object classes in the Alfabet database. ADIF provides a mechanism that handles role and indicator import for an object class without the need to define all required references separately in the configuration of the ADIF scheme.

The following information is available about import of references:

- [Import of References and ReferenceArrays with Reference Support Set to False](#)

- [Import of ReferenceArrays with Reference Support Set to True](#)

- [Defining a Role for Imported Objects](#)

- [Defining an Indicator for Imported Objects](#)

Import of References and ReferenceArrays with Reference Support Set to False

The information about references is stored in the database table of the referencing object class. The database column of the referencing object class property must contain the REFSTR property value of the referenced object class.

The REFSTR is a unique property specific for the Alfabet meta-model and usually not part of the external data. Referenced objects are usually specified via key attributes, like the name of the referenced object. The import of values for a property of the type Reference therefore normally requires that the REFSTR of the imported object is written to the temporary table prior to data import to the Alfabet database via an SQL command in the folder **SQL Commands - OnStart**.

For example, the import of device data shall include the specification of an authorized user. The object class Device in the Alfabet database has a property ResponsibleUser that defines the responsible user via a reference to the object class Person. The required value to fill the database column of the property ResponsibleUser is the REFSTR of the object class person.

In the import file, the authorized user is defined by the user's login name, corresponding to the USER_NAME property of the object class person. The column name in the external data table is Owner:

	A	B	C	D	E
1	Name	Version	StartDate	End Date	Owner
2	ImportDevice3	2.0	01.04.2011	18.05.2018	Ngombe
3	ImportDevice4	3.0	24.08.2011	31.12.2014	Picard

When defining the import entry triggering import, perform the following configurations in the import entry to set the values for the property ResponsibleUser of the object class Device:

1) Define an **Attribute** element that writes the data about the users login name to the temporary table. Do not map the column to a property of the target database table in the Alfabet database.

2) Define an **Attribute** element that creates a temporary database column not filled with external data during import. Map the column to the target object class property establishing the relation. In the example, the empty column is mapped to the property ResponsibleUser of the target object class Device in the Alfabet database.

3) Right-click the folder **SQL Commands - OnStart** and select **Create SQL Command**.

4) Click the new SQL command and specify the following in the attribute window:

- **Name**: Enter a meaningful name for the SQL command.

- **Comments**: Enter a comment that provides information about the functionality implemented by the SQL command.

- **Command Type** : Select `DMLStatement`.

- **Result Type** : Select `Undefined`.

- **Ignore Errors** : Select `True` if you want the export to be executed even if the SQL statement specified with the attribute **Text** results in an exception. Select `False` if you do not want the export to be executed in case of an error in the SQL statement.

- **Is Active** : Select `True` to activate the SQL command. Select `False` to deactivate the SQL command.

- In the attribute window, click the **Browse** button in the **Text** attribute to open an editor for the definition of the SQL query. Define the SQL query in the **SQL Text** tab of the editor.

The SQL query must be an `UPDATE` statement targeting the temporary database table. The query must fill the column for the property establishing the relation with the `REFSTR` of the referenced objects:

```
UPDATE TMP_DEVICE

SET OWNER_REF = (SELECT p.REFSTR FROM PERSON p WHERE p.USER_NAME =
TMP_DEVICE.OWNER)
```

Import of ReferenceArrays with Reference Support Set to True

Relations of the type `ReferenceArray` are stored in the `RELATIONS` table. The ADIF interface provides a mechanism to fill the `RELATIONS` table with the required data during data import to a target object class.

It is not allowed to alter the `RELATIONS` table directly via an SQL command entering data from a temporary table into the `RELATIONS` table. To ensure data integrity, `RELATIONS` of the type `ReferenceArray` with attribute **Reference Support** set to `True` must be imported with the mechanisms described below.

The temporary table must match the following criteria to import reference array entries to the `RELATIONS` table:

- Both the `REFSTR` of the referencing object and the `REFSTR` of the referenced object can be read from the temporary table defined in the import entry.

- For each relation to be added to the `RELATIONS` table (for each combination between referencing object and referenced object), there must be a row in the temporary table.

The `REFSTR` is a unique property specific to the Alfabet meta-model and usually not part of the external data. Referenced objects are usually specified via key attributes, like the name of the referenced object. The import of values for a property of the type `Reference` therefore normally requires that the `REFSTR` of the imported object is written to the temporary table prior to data import in the Alfabet database via an SQL command in the folder **SQL Commands - OnStart**.

- Define an **Attribute** ⬜ element that writes the data about the users login name to the temporary table. Do not map the column to a property of the target database table in the Alfabet database.

- Define an **Attribute** ⬜ element that creates a temporary database column not filled with external data during import. Map the column to the target object class property establishing the relation. In the example, the empty column is mapped to the property `ResponsibleUser` of the target object class `Device` in the Alfabet database.

- Define an SQL command ⚡ in the folder **SQL Commands - OnStart** and define an `UPDATE` statement executed on the temporary table that fills the empty column with the `REFSTR` of the referenced object.

To trigger the changes in the `RELATIONS` table, you must add a relation ⬛ element to the import entry:

1) Right-click the **Relations** folder of the import entry 🖼 for which you want to import the reference array data and select **Create Relation**. The new relation ⬛ element is added as a sub-node of the **Relations** folder.

2) Click the relation ⬛ element and specify the following attributes in the attribute window:

   - **From Class**: in the drop-down list, select the object class of the Alfabet database containing the referencing property.

   - **Relation Name**: Enter the name of the referencing property.

   - **From Column**: Enter the name of the column of your temporary table that stores the `REFSTR` of the referencing object.

   - **To Class**: In the drop-down list, select the referenced object class of the Alfabet database.

   - **To Column**: Enter the name of the column of your temporary table that stores the `REFSTR` of the referenced object.

   - **Clear Before Create**: Select `True` if you want all relations established by the property defined with the attribute **Relation Name** to be deleted from the Alfabet `RELATIONS` table prior to import of the new relations. Select `False` if you want to write new relations to the `RELATIONS` table without clearing the `RELATIONS` table.

   👁 For example, data about devices and information about the assignment of the devices to device groups shall be imported via an XML file. In the XML file, each device group is represented by an element `DeviceGroups`. The devices in the device group are represented by child elements `Device` in the respective `DeviceGroup` elements:

```
<Groups>

    <DeviceGroup Name="ImportGroup1">

        <Description>Server hosts</Description>

        <Device Name="ImportDevice1" Version="1.0"
        StartDate="2011.01.01" EndDate="2015.01.05"/>

        <Device Name="ImportDevice2" Version="1.2"
        StartDate="2011.08.24" EndDate="2014.12.31"/>

    </DeviceGroup>

    <DeviceGroup Name="ImportGroup2">

        <Description>Web relevant devices</Description>
```

```
        <Device Name="Backup Server" Version="2.2.1"
        StartDate="2009.09.30" EndDate="2013.09.01"/>

        <Device Name="ImportDevice1" Version="1.0"
        StartDate="2011.01.01" EndDate="2015.01.05"/>

    </DeviceGroup>

</Groups>
```

In the ADIF import scheme for import of the data, one import entry is added for the import of device groups and one for import of devices. The relation between the device groups and devices is stored via the automatically generated attribute `DeviceGroup_Id`.

(For details about processing the XML data, see *Defining Data Import From XML Files in an XML Import Set*.)

In the Alfabet meta-model, the relation between device groups and devices is defined via the object class `Device` with the property `DeviceGroups` and via the object class `DeviceGroup` with the property `Devices`. In this example, the property `DeviceGroups` of the object class `Device` is used to define the relation. The back reference will be automatically set by the import mechanism during data import.

To establish a relation, the ADIF import mechanism has to find both the `REFSTR` of the referencing object and the `REFSTR` of the referenced object in the temporary database table for which the relation element is specified. The relation element is processed after the import of data into the database table of the target object class in the Alfabet database and therefore the `REFSTR` of the referencing device can be read from the automatically generated `REFSTR` column of the temporary table.

The `REFSTR` of the device group must be added to the temporary table via an SQL command of the type `OnStart`. Therefore, the order of import entries in the ADIF scheme is configured to trigger the import of device group data prior to the import of device data. This is required to ensure that references are only set to objects already existing in the Alfabet database and provides information about the `REFSTR` of the device groups in parallel to the information about the `DeviceGroup_Id` in the temporary table for device groups.
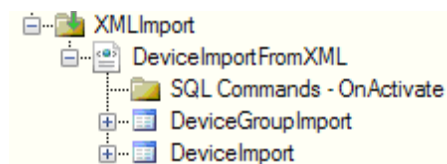
When defining the temporary table for the import of device data, an empty database column is added to the temporary table that is neither filled with external data nor mapped to the target Alfabet database table.

Import Entry for DeviceGroup — DeviceImportFromXML
- SQL Commands - OnActivate
- DeviceGroupImport
  - Attributes
    - Name -> NAME -> Name
    - DeviceGroup_Id -> DEVICEGROUP_ID -> None
    - Description -> DESCRIPTION -> Description
  - Relations
  - Indexes
  - SQL Commands - OnActivate
  - SQL Commands - OnStart
  - SQL Commands - DataBind
  - SQL Commands - OnComplete

Columns storing hierarchy information

Import Entry for Device — DeviceImport
- Attributes
  - Name -> NAME -> Name
  - Version -> VERSION -> Version
  - StartDate -> STARTDATE -> StartDate
  - EndDate -> ENDDATE -> EndDate
  - DeviceGroup_Id -> DEVICEGROUP_ID -> None
  - None -> DEVICEGROUP -> None
- Relations
- Indexes
- SQL Commands - OnActivate
- SQL Commands - OnStart
- SQL Commands - DataBind
- SQL Commands - OnComplete

Empty column for REFSTR of device group

An SQL command is added to the folder **SQL Commands - OnStart** that fills the empty column DEVICEGROUP with the REFSTR of the parent device group using the DeviceGroup_Id as a mapping property:

```
UPDATE TMP_DEVICE

SET DEVICEGROUP =(SELECT REFSTR FROM TMP_DVG

WHERE TMP_DVG.DEVICEGROUP_ID = TMP_DEVICE.DEVICEGROUP_ID)
```

This is how the temporary table that results looks:

	NAME	VERSION	STARTDATE	ENDDATE	DEVICEGROUP_ID	DEVICEGROUP	REFSTR
1	ImportDevice1	1.0	01/01/2011 00:00:00 0	06/01/2015 00:00:00 0	0	96-66-0	20-129-0
2	ImportDevice2	1.2	24/08/2011 00:00:00 0	31/12/2014 00:00:00 0	0	96-66-0	20-130-0
3	Backup Server	2.2.1	30/09/2009 00:00:00 0	01/09/2013 00:00:00 0	1	96-67-0	20-6-0
4	ImportDevice1	1.0	01/01/2011 00:00:00 0	05/01/2015 00:00:00 0	1	96-67-0	20-129-0

The table has one row for each combination of device and device group. That means that there are two rows for the device ImportDevice1 that is assigned to two device groups. Nevertheless data bind has processed this information correctly. As a result, the column REFSTR displays the same REFSTR in both lines.

A relation element has been added to the **Relations** folder of the import entry for the device import. The data in the columns DEVICEGROUP and the REFSTR of the temporary table are used to define the relation:
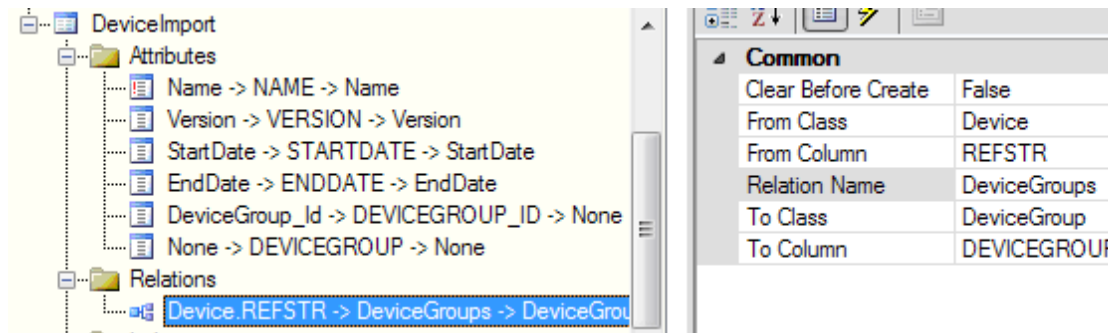
Defining a Role for Imported Objects

An object of the object class Role is created each time a responsibility is defined for an object based on a preconfigured role type.

Role types define the type of responsibility a user or organization has for an object in the Alfabet database. Role types are configured in the **Reference Data** functionality of the **Configuration** module of Alfabet. They are then assigned to object classes in the **Class Configuration** functionality.

In the **Responsibilities** page view of an object, the user can now create a role by defining a responsibility for either a user or an organization. In the editor for the role definition, the responsible user or organization and the role type that defines the type of responsibility for the object is defined.

In the Alfabet class model, the relation between the role type, the object, and the responsible person or organization is only specified in the database table for the class Role. The information is not including any object class properties of the type Reference or ReferenceArray of the object class Person or OrgaUnit or the object class the role is defined for:
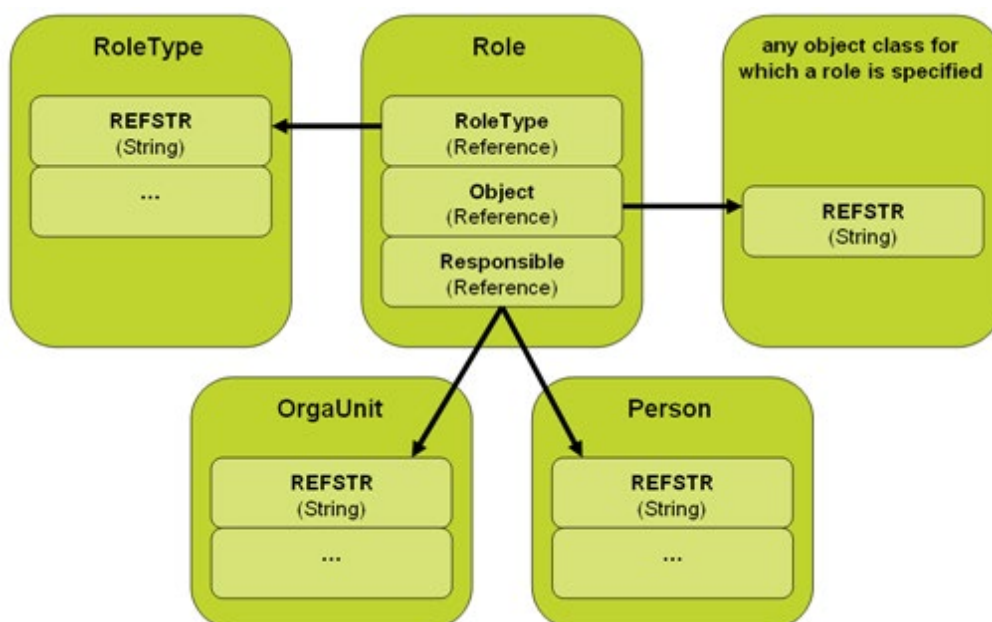
To include information about roles for an imported object class in the ADIF import, you do not have to additionally include import of the `Role` object into your ADIF scheme via a separate ADIF import entry. Instead, the creation of the `Role` object can be performed automatically during import of objects of the object class for that the role is set with the following configuration:

1) Define an **Attribute** ⬚ element that writes a unique property of the user (object class `Person`) or organization (object class `OrgaUnit`) that shall have responsibility for the object via the role to the temporary table. Usually this will be the name of the user or organization. Do not map the column to a property of the target database table in the Alfabet database.
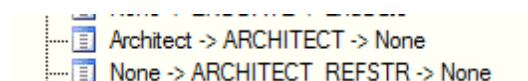
> ⓘ For information about the definition of **Attribute** ⬚ elements see Creating Attribute Entries to Map the Database Table Columns.

2) Define an **Attribute** ⬚ element that creates a temporary database column neither filled with external data during import nor mapped to a target object class property. In a later step, the column is filled via an SQL query with the `REFSTR` of the user or organization.

3) Right-click the folder **SQL Commands - OnStart** and select **Create SQL Command**.

4) Click the new SQL command 🗲 and specify the following in the attribute window:

   • **Name**: Enter a meaningful name for the SQL command.

   • **Comments**: Enter a comment that provides information about the functionality implemented by the SQL command. The SQL command is used to enter the `REFSTR` of the user or organization responsible for the object via the role into the temporary database table.

   • **Command Type**: Select `DMLStatement`.

   • **Result Type**: Select `Undefined`.

   • **Ignore Errors**: Select `True` if you want the export to be executed even if the SQL statement specified with the attribute **Text** results in an exception. Select `False` if you do not want the export to be executed in case of an error in the SQL statement.

   • **Is Active**: Select `True` to activate the SQL command.

   • In the attribute window, click the **Browse** ⬚ button in the **Text** attribute to open an editor for the definition of the SQL query. Define the SQL query in the **SQL Text** tab of the editor.

The SQL query must be an `UPDATE` statement targeting the temporary database table. The query must fill the column for the `REFSTR` of the user or organization responsible for the object via the role with the `REFSTR` of the user or organization.

> 👁 In the following example, the role Architect is defined for imported applications. In the Attribute folder of the import entry for the object class `Application`, two **Attribute** ⬚ elements have been defined for the import. The `TECH_NAME` property of the object class `Person` is read into a database column `ARCHITECT` during import without further import to a standard object class table of the Alfabet database. The additional column `ARCHITECT_REFSTR` is defined for the temporary table without defining data import or data integration into a standard object class table of the Alfabet database.
>
> 📄 Architect -> ARCHITECT -> None
> 📄 None -> ARCHITECT_REFSTR -> None

The following SQL command fills this column with the `REFSTR` of the user identified via the technical name (property `TECH_NAME`) in the column `ARCHITECT` of the temporary table:

```
UPDATE TMP_APPLICATION

SET ARCHITECT_REFSTR = (SELECT REFSTR FROM PERSON WHERE
PERSON.TECH_NAME = TMP_APPLICATION.ARCHITECT)
```

5) Right-click the folder **Roles** and select **Create Role**.

6) Click the new role and specify the following in the attribute window:

- **Class Name**: Enter `Person`, if the role is assigned to a user or `OrgaUnit`, if the role is assigned to an organization.

- **Database Column**: Enter the name of the database column of the temporary table that will be filled with the `REFSTR` of the user or organization the role is assigned to.

- **Role Type Name**: Enter the name of the role type the role is defined for. This is the value of the object class property `Name` of the object of the object class `RoleType` assigned to the role.

For the example above, the required definition of the role element is:
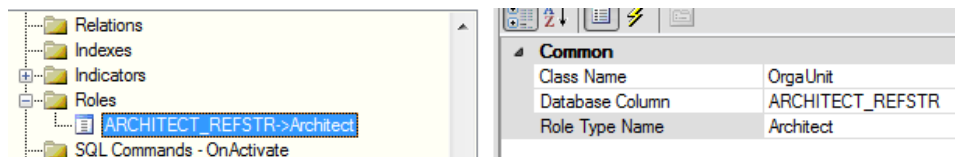
Defining an Indicator for Imported Objects

In the **Evaluations and Portfolios** functionality in the **Configuration** module in Alfabet, you can define evaluation types that bundle a set of indicator types in order to evaluate objects in Alfabet. The evaluation types are assigned to object classes in the **Class Configuration** functionality.

The indicator types defined for the evaluation types that are assigned to the object class are displayed on the *Evaluation Page View* of each object in the object class. The user can define indicators for the object by defining the value of the indicator type in the *Evaluation Page View*.

An object of the object class Indicator is created each time a value is entered for an indicator type on the **Evaluations** page view of an object.

In the Alfabet class model, the value defined for an object's indicator type is only specified in the database table for the class `Indicator`. The class `Indicator` specifies the object and the indicator type that each value was defined for and the evaluation type that the indicator type is assigned to:
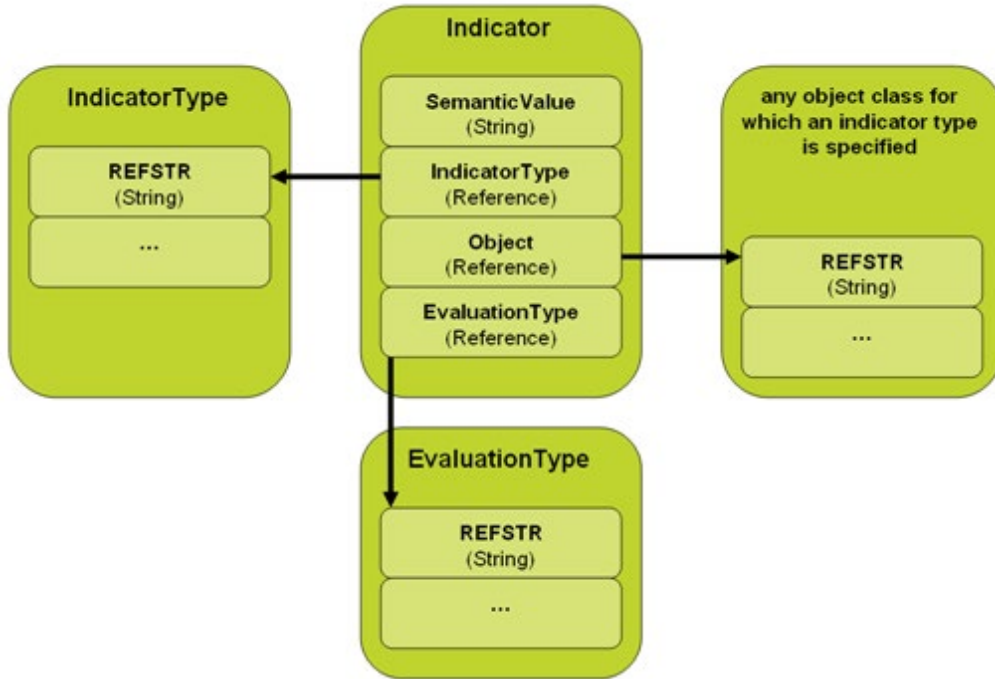
To include information about indicators for an imported object class in the ADIF import, you do not have to additionally include import of the `Indicator` object into your ADIF scheme via a separate ADIF import entry. Instead, the creation of the `Indicator` object can be performed automatically during import of objects of the object class for that the role is set with the following configuration:

1) Define an **Attribute** ⊡ element that writes the values for indicators that shall be set for the imported objects into a column of the temporary database table. This value must have the format required for the object class `SemanticValue` of the object class Indicator and correspond to the configuration of allowed indicator values resulting from the indicator type and evaluation type definitions. Do not map the column to a property of the target database table in the Alfabet database.

> ⓘ For information about the definition of **Attribute** ⊡ elements see Creating Attribute Entries to Map the Database Table Columns.
>
> For information about the configuration of indicator types and evaluation types see the chapter *Configuring Evaluations, Prioritization Schemes, and Portfolios* in the reference manual *Configuring Alfabet with Alfabet Expand*.

2) Right-click the folder **Indicators** and select **Create Indicator**.

3) Click the new indicator ⊡ and specify the following in the attribute window:

   • **Database Column**: Enter the name of the database column of the temporary table that will be filled with the value for the indicator during import.

   • **Indicator Type Technical Name**: Enter the values of the object class property `TechnicalName` of the `EvaluationType` assigned to the indicator followed by | followed by the values of the object class property `TechnicalName` of the `IndicatorType` assigned to the indicator (EvaluationTypeTechnicalName|IndicatorTypeTechnicalName). If the indicator shall not be assigned to an evaluation type, you can specify the technical name of the indicator type (`IndicatorType`) only (IndicatorTypeTechnicalName).

Collecting Data from Multiple External Files or Database Tables in one Temporary Table

Standard data import leads to the creation of one temporary table for the import of data from each external file or database table. If you want to import data from multiple files with the same data structure, but different content, you can configure the ADIF import scheme to write data from multiple files / external database tables to a single temporary table. This method enhances the performance of the import process, because fewer tables must be created and deleted from the database and the mapping of the imported data records to the data records that already exist in the Alfabet database must only be executed once.
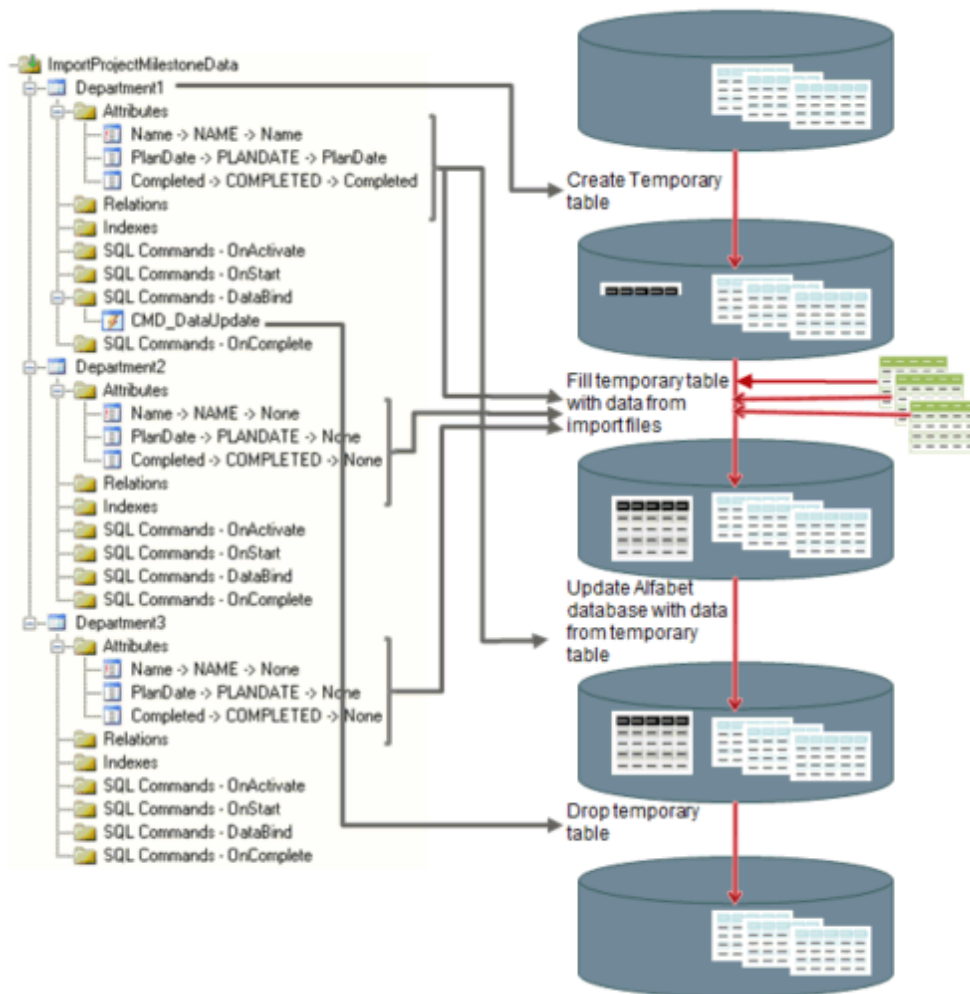
Configure the following to process data from multiple import sources in one temporary table during import:

- Create an **Import Entry** and configure a complete data import to the target Alfabet database table via the temporary table in the import entry. This requires that you specify at least the attribute elements mapping data between the import data, the temporary table and the Alfabet database table, and that you set the attribute **Table Owner** of this import entry to `True`. This import entry is called the table owner in the following.
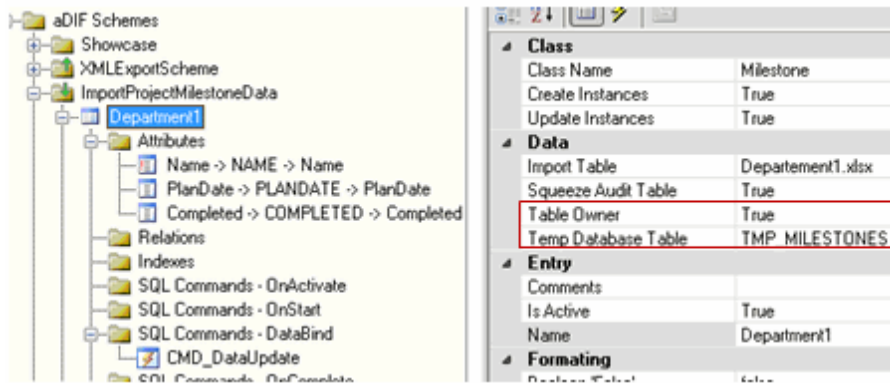
- For each additional external import file/database table, create an **Import Entry** 📇 that defines the import of the data to the temporary table defined in the table owner import entry:

  - The attribute **Temp Database Table** must be identical with the **Temp Database Table** specified for the table owner.

  - The attribute **Table Owner** must be set to `False`.

  - The **Attribute** 📄 elements must specify the import of data to the temporary table only. The import to the Alfabet database table must be left undefined.

  - Data bind must not be defined.

No definition of planningIT target database table

No data bind definition
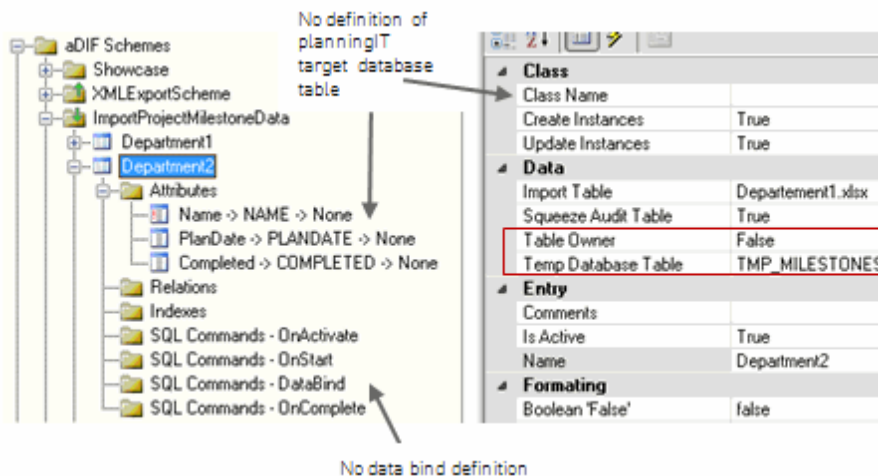
Semi-Automatic Creation of Import Entry Definitions for Import Files

The ADIF configuration interface offers a mechanism that reads an import file and automatically creates import entries and **Attribute** 📄 elements for the import entry matching the data in an import file. The automatically created entries are incomplete, because they can not cover the definition of import from the temporary database table to the target Alfabet database table.

⚠️ During automatic pre-configuration of the import entry, the name for the temporary table is defined as `TMP_`<import file name without extension> and the name of the database columns for the temporary table are derived from the column headers of the import file. The mechanism

substitutes white spaces and slashes with underscores, but other special characters in the file name or column header names will be ignored or may even lead to invalid technical names for database tables or columns.

You can either define a single import entry for an import from a single Microsoft® Excel® or comma-separated format file or a file set for the import from multiple import files. For the import from XML files, a file set covering import from all available elements in the XML can also be created via the mechanism.

Semi-Automatic Creation of an Import Entry for a Single Import File

To use this mechanism, the import file must be a Microsoft® Excel®, CVS or TXT file and must be available in the local file system.

1) In the explorer of the ADIF configuration interface, right-click the ADIF import scheme or the file import set that you want to create an import entry for and select **Create Entry From File**. An explorer window opens.

2) In the explorer, select the import file format in the drop-down list next to the field for the file name.
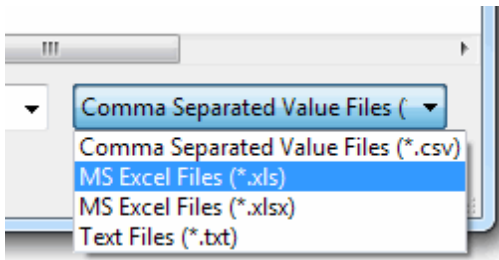
3) Navigate to the import file in the local file system and click **Open**. The new import entry is added to the ADIF import scheme with **Attribute** elements that correspond to the import data automatically added to the import entry

4) Review the names generated for the temporary table and the columns of the temporary tables for technical correctness and uniqueness.

5) Complete and modify the automatically generated definition. For details, see: *Defining Import of External Data in an Import Entry*.

Semi-Automatic Creation of Multiple Import Entries in a File Import Set

To use this mechanism, the import files must be Microsoft® Excel®, CVS or TXT files. All files for which import entries shall be created must be located in the same directory and the directory must not contain other files.

1) In the explorer of the ADIF configuration interface, right-click the ADIF import scheme that you want to create an import set for and select **Create File Set from Directory**. A selector window opens.

2) In the selector, select the directory containing the import files and click **OK**. The new import set is added to the ADIF import scheme with import entries including child **Attribute** elements matching the import data.

3) Review the names generated for the temporary table and the columns of the temporary tables for technical correctness and uniqueness.

4) Complete and modify the automatically generated definition.

5) For details, see: *Defining Import of External Data in an Import Entry*.

Semi-Automatic Creation of an XML Import Set from an XML File

To use this mechanism, the import file must be a valid XML file and must be available via the local file system.

1) In the explorer of the ADIF configuration interface, right-click the ADIF import scheme that you want to create import entries for and select **Create XML Set from File**. A selector window opens.

2) In the selector, select the directory containing the import files and click **OK**. The new import set is added to the ADIF import scheme with import entries 🔲 including child **Attribute** 🔲 elements matching the import data.

3) Review the names generated for the temporary table and the columns of the temporary tables for technical correctness and uniqueness.

4) Complete and modify the automatically generated definition.

5) For details, see: *Defining Import of External Data in an Import Entry*.

Configuring SQL Commands for Optional Enhanced Import Features

The basic import configuration can be enhanced by the definition of additional SQL commands. When creating ADIF import schemes via the ADIF configuration explorer, folders for SQL commands are automatically added to the explorer if SQL commands can be defined for an element. The folders also indicate the SQL command type.

ⓘ In the XML of the ADIF import scheme each SQL command element has an attribute **Type**. For each allowed command type, a separate folder named **SQL Commands - <Type>** is added to the explorer. When SQL commands are defined in a folder, the **Type** attribute is automatically set in the XML definition. When specifying SQL commands in an XML editor, the **Type** attribute must be defined manually.

The following SQL command types are available:

Type	Location in ADIF Scheme	Description
OnActivate	Import sets and import entries	SQL commands of the type `OnActivate` define conditions for the execution of the element they are defined for.

Type	Location in ADIF Scheme	Description
		For more information, see *Configuring the Conditional Execution of Parts of the Import Scheme*. **Note**: Changes triggered by OnActivate commands are not rolled back if the option **Commit After Run** is set to `False` for the import scheme.
DataBind	Import entries	SQL commands of the type `DataBind` define how data in the temporary tables are mapped with the data in the target Alfabet database table. The definition of data bind commands is optional. Alternatively, the conditions for data mapping can be defined in the **Attribute** elements of the import entry via the attribute **Is Key**. For more information, see *Mapping External Data to Temporary Tables and Alfabet Database Tables*.
DataUpload	Import entries	SQL commands of the type `DataUpload` are required for data import from external databases and active directories. The query is executed on the external source. The output of the query must be a tabular data set that is the external data that will be imported. For more information, see *Defining Data Upload from an External Database in a Database Import Set* or *Defining Data Upload from an LDAP Table in an LDAP Import Set*.
OnStart	Import entries	SQL commands of the type `OnStart` are executed after data is written into temporary database tables and prior to the SQL commands of the type `DataBind`. SQL commands of the type `OnComplete` are executed after the SQL commands of the type `DataBind`. The commands allow special operations to be executed on the Alfabet database prior or after import of data to the target Alfabet database tables. The SQL commands can be defined for the following purposes: • Definition of customized debug information written to the log file during execution of imports. • This configuration is described in the section *Configuring Logging Parameters*. • Manipulation of the Alfabet database or the temporary database table prior and/or after data import. The definition of SQL command for this purpose is described in this section.
OnComplete	Import entries	

One of the main features of the ADIF interface is the flexibility in configuration. The ADIF import scheme can not only be configured to simply read data from an external file and add it to the Alfabet database, but you can configure ADIF to perform additional operations on the Alfabet database, the temporary database

tables created during import, or, in case of import from an external database, on the external database table at different stages of the processing of the ADIF import scheme. SQL queries of the type DLL statement or DML statement can be defined in any number and order required.

> ⓘ If SQL queries are not sufficient to implement the desired import, custom DLLs can be ordered from Software AG to provide functionality. The custom code is added to the ADIF import scheme via an SQL command with the **CommandType** StoredProcedure. The correct setting of configuration elements and attributes depends on the custom code. A description is delivered with the code on an individual basis.
>
> StoredProcedures are executed in the same way as DML statements (iIn the order of the execution of SQL statements as described below).

When configuring SQL commands for an ADIF import scheme, the order of execution must be taken into account. The elements of an ADIF scheme are executed in the following order:

- The SQL commands OnActivate are evaluated first and the respective elements of the ADIF schemes are included or excluded from the execution schedule.

- All SQL commands of the type OnStart and the **Command Type** DDLStatement that target the Alfabet database are executed in the order they are specified in the ADIF import scheme.

- Data is uploaded to the temporary tables for all import entries in the order specified in the ADIF import scheme. For import from database import sets and LDAP import sets, data import to the temporary tables is associated with the following processing steps:

  - The connection to the external database is established.

  - All DDL statements of SQL commands of the type OnStart targeting the external database configured for the import entries in the import set are executed in the order they are defined in the import set.

  - The import entries are processed in the order they are configured within the database import set as follows:

    - All DML statements of SQL commands of the type OnStart targeting the external database are executed in the order they are specified in the import entry 🗒.

    - The SQL command of the type DataUpload is executed, and data is uploaded to the temporary table.

    - All DML statements of SQL commands of the type OnComplete targeting the external database are executed in the order they are specified in the import entry 🗒.

  - All DDL statements of SQL commands of the type OnComplete targeting the external database configured for the import entries in the import set are executed in the order they are defined in the import set.

- All import entries are processed as follows independent of the location inside or outside an import set:

  - All DML statements of SQL commands of the type OnStart are executed in the order they are specified in the import entry 🗒.

  - Data is imported to the target Alfabet database tables executing **Attribute** 🗒 elements, SQL commands of the type DataBind and automatic data mapping mechanisms of the ADIF import process.

- Relation elements are processed.

- The DML statements of SQL commands of the type `OnComplete` are executed in the order they are specified in the import entry 🗔 .

- All DDL statements of SQL commands of the type `OnComplete` targeting the Alfabet database configured for the export entries in the ADIF export scheme are executed in the order they are defined in the ADIF export scheme.

This section describes in general how to define an SQL command within an **SQL Commands - OnStart** or **SQL Commands - OnComplete** folder of the ADIF configuration explorer and describes two possible use cases for the definition of SQL commands:

- *Use Case: Filling Empty Property Values with Default Values When Importing to Mandatory Object Class Properties*

- *Use Case: Defining Import-Related Custom Properties*

Creating SQL Commands

To specify an SQL command to be executed on the Alfabet database or an external target database:

1) In the explorer of the ADIF configuration interface, right-click the folder **SQL Commands - OnStart** or **SQL Commands - OnComplete** that is automatically added to the explorer when an import entry 🗔 is created.

2) In the context menu, select **Create SQL Command**. A new SQL command 🗲 is added as a child node of the folder.

3) Click the new SQL command 🗲 and specify the following in the attribute window:

   - **Name**: Enter a meaningful name for the SQL command.

   - **Comments**: Enter a comment that provides information about the functionality implemented by the SQL command.

   - **Command Type**: In the drop-down list, select the type of SQL query that you want to define. Select either:

     - `DMLStatement` for SQL data manipulation statements like `SELECT, INSERT, UPDATE, DELETE, MERGE`.

     - `DDLStatement` for SQL data definition statements like `CREATE, ALTER, DROP, TRUNCATE, RENAME`.

       ⚠️
       - DDL statements are only permitted to create, alter and drop new database tables. It is not permissible to apply those commands to standard Alfabet database tables.

       - The ADIF import mechanisms are designed to provide a high level of security. Always check whether an import can be performed via an import mechanism provided by ADIF prior to defining your own SQL commands to perform the import.

- **Result Type** : Select `Undefined`.

- **ApplyTo**: If the SQL command is defined for import from an external database table, define the database that the SQL command is executed on:

  - local for execution on the Alfabet database

  - external for execution on the external database

    ⚠️ The execution of SQL commands on external database tables is not recommended for the data import. External databases should be manipulated directly in the source database only.

- **Ignore Errors** : Select `True` if you want the export to be executed even if the SQL statement specified with the attribute **Text** results in an exception. Select `False` if you do not want the export to be executed in case of an error in the SQL statement.

- **Is Active** : Select `True` to activate the SQL command. Select `False` to deactivate the SQL command.

- **Text**: Define the query for the command. Either write the query directly in the attribute field or click the **Browse** ⊡ button to open an editor for the definition of the query in the **SQL Text** tab of the editor.

  ⓘ The queries to be executed on external database must match the syntax required on the external database/LDAP table.

  ⓘ The text editor for the definition of queries in the ADIF configuration interface provides help for the definition of the query in separate tabs. For more information see *Defining SQL Queries for SQL Commands* in the chapter *Configuring ADIF Schemes*.

Use Case: Filling Empty Property Values with Default Values When Importing to Mandatory Object Class Properties

Some object class properties of object classes in the Alfabet meta-model are mandatory and must be defined when new data records are added. You can identify mandatory properties in the **Meta-Model** subtree of the ADIF configuration explorer by their yellow/red colouring:
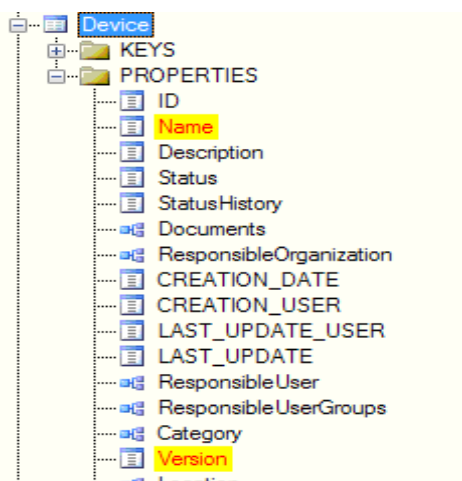
When creating new records in the Alfabet database during import, the mandatory information must be imported for all new records. If the external data set that is imported does not provide the required information, the required values must be written via SQL commands to the temporary table.

During import, the data is first written to the temporary table. Then SQL commands of the type `OnStart` are executed prior to import of the data in the temporary table to the target database table of the Alfabet database.

An SQL command `OnStart` can be defined that updates the temporary table, for example, by setting a default value for all empty fields in the column for the mandatory property.

For example, an import file containing data about devices does not provide version numbers for all devices. The `Version` property is a mandatory property of the object class `Device`.

Mandatory Version information is not provided consistently

	A	B	C	D
1	Name	Version	StartDate	End Date
2	ImportDevice1	4.5.9	02.01.2010	15.03.2015
3	ImportDevice2		15.06.2010	30.03.2012
4	ImportDevice3	2.0	01.04.2011	18.05.2018
5	ImportDevice4	3.1	24.08.2011	31.12.2014

To add the missing version information to the temporary table resulting from import of the data displayed above, the following SQL query can be defined in the **Text** attribute of an SQL command of the type `OnStart`. The query sets the version to 1.0 for all devices for which no version information is available in the external data:

```
UPDATE TMP_DEVICE

SET Version ='1.0'

WHERE Version = NULL;
```

Use Case: Defining Import-Related Custom Properties

It may be useful to store the information about the import directly in the Alfabet database table of the target objects. This may be useful, for example, for the following reasons:

- Configure import to only include objects that are not already marked as imported in a prior data import.

- Inform the user via the Alfabet interface whether and/or when data has been last imported.

- Use the attribute to define a workflow template that starts a workflow only if the attribute is set to a defined value. The ADIF import scheme can additionally be configured to start the workflows automatically as part of the import process.

For more information see *Configuring the Automatic Start of Workflows During Import*.

- Use the attribute as a condition for the execution of actions via the Alfabet interface (for example, in workflow pre-conditions or in reports that only shows data that was imported via ADIF).

The custom property may contain the date the import was performed or a boolean value that distinguishes between imported and not imported data.

The following configuration steps are required to store information about the import directly in the database table of the imported objects:

- A custom property must be added to the object class for which data is imported. Custom properties must be configured in the configuration tool Alfabet Expand. For information about how to create a custom property, see the section *Configuring Custom Properties for Protected or Public Object Classes* in the reference manual *Configuring Alfabet with Alfabet Expand*.

- In the import entry ⊞ triggering the import, define an empty database column in the temporary database table and map it to the custom property of the target Alfabet database table.

- Add an SQL command 🗒 with an `UPDATE` statement in the folder **SQL Commands - OnStart** that writes the information about the import to the column for the custom property in the temporary table.

You must use the configuration capabilities in Alfabet Expand to add new property columns to a database table of a Alfabet object class. It is not permissible to add a property column via a DDL statement in the SQL commands of the ADIF scheme. **Existing standard Alfabet database tables must not be altered by SQL commands via ADIF.**

Configuring Execution of the Import Scheme Dependent on Current Parameters

ADIF import schemes can be configured to adapt to current states of objects in the Alfabet database. You can define conditions for the overall execution of parts of the ADIF import scheme or define conditions in the command line of the ADIF console application that specify the choice of object data for execution, which vary per import.

Configuring the Conditional Execution of Parts of the Import Scheme

The ADIF import scheme allows you to tie conditions to the execution of parts of the import definition. You can define SQL commands that check the Alfabet database for the availability of data and configure import sets or import entries to be executed only if a configured SQL query executed on the Alfabet database delivers either a positive or negative result.

Conditions can be defined for the following elements of the ADIF import scheme:

- Database import set 🛢

- File import set 📄

- XML import set

- Import entry

If a condition is specified for an element, this element and all child elements thereof are only executed if the condition is met.

To specify a condition for execution of a part of an ADIF import scheme:

1) In the explorer of the ADIF configuration interface, right-click the folder **SQL Commands - OnActivate** that is automatically added to the explorer on creation of an element.

2) In the context menu, select **Create SQL Command**. A new SQL command is added as a child node of the **SQL Commands - OnActivate** folder.

3) Click the new SQL command and specify the following attributes in the attribute window:

   - **Name**: Enter a meaningful name for the SQL command.

   - **Comments**: Enter a comment that provides information about the functionality implemented by the SQL command.

   - **Apply to**: Select `local`. SQL commands of the type **OnActivate** can only be executed on the Alfabet database.

   - **Result Type**: In the drop-down list, select one of the following:

     - `PositiveCheck`: If the data set that results from the execution of the query defined in the **Text** attribute is empty, the current element of the ADIF scheme will not be executed and the message defined in the **Message** attribute will be written to the log file.

     - `NegativeCheck`: If the data set that results from the execution of the query defined via the **Text** attribute is not empty, the current element of the ADIF scheme will not be executed and the message defined in the **Message** attribute will be written to the log file.

   - **Text**: Define an SQL query with a `SELECT` statement that returns a data set. The resulting data set is checked according to the settings of the attribute **Result Type** in order to specify whether the ADIF element is executed. Either write the SQL query directly in the attribute field or click the **Browse** button to open an editor for the definition of the SQL query. Define the SQL query in the **SQL Text** tab of the editor.

   - **Message**: Enter a text that shall be written to the log file in case the ADIF element is not executed.

   - **Ignore Errors**: Select `True` if you want the import to be executed even if the SQL statement specified with the attribute **Text** results in an exception. Select `False` if you do not want the import to be executed if an error in the SQL statement occurs.

   - **Is Active**: Select `True` to activate the check. Select `False` to deactivate the check.

The condition for the check can be defined in conjunction with parameters set during runtime of the ADIF import.

For more information, see *Configuring Logging Parameters*.

Configuring Import Dependent on Parameters

Import can be configured to be based on parameters defined for example in the command line of the ADIF console application or in the JSON body of the RESTful service call when starting an import. This allows you to reuse an ADIF import scheme for various similar imports that only vary in small aspects.

> (i) It is not possible to define parameters for ADIF imports executed via the user interface. Nevertheless, these imports can be executed with the default values for the parameters if these default values are defined in the ADIF import scheme. If an ADIF scheme configuration includes mandatory parameters, the execution option in the **Adif Jobs Administration** (`ADMIN_Adi-fJobs`) functionality.

The following information is available:

- [Configuring an ADIF Import Scheme to Use Parameters in Import Definitions](#)

- [Defining Parameter Values On Execution of ADIF Schemes](#)

- [Defining Parameter Values During Testing of the ADIF Scheme](#)

Configuring an ADIF Import Scheme to Use Parameters in Import Definitions

Parameters have to be defined in the ADIF scheme. Only parameters that are defined in the ADIF scheme can be assigned a value during execution of the ADIF scheme.

To define the parameters for an ADIF scheme:

1) In the explorer, click on the ADIF import scheme that you would like to start with parameters.

2) In the attribute window, define the parameter relevant attributes:

   - **Parameters Backward Compatibility Mode**: Make sure that the parameter is set to `False`. This is the default value for new ADIF schemes.

     > (i) ADIF schemes defined prior to Alfabet 10.4 will have the compatibility mode set to `True`. If you reset the compatibility mode for these schemes, parameter definitions must be added to the ADIF import entries in the scheme and the way parameters are defined in the SQL commands must be revised.
     >
     > Please note that ADIF schemes with a **Parameters Backward Compatibility Mode** set to `True` which contain parameter definitions cannot be scheduled for execution via the **Job Schedule** functionality.

   - **Arguments Table Name**: At the start of an ADIF import, the parameter values are written to a temporary database table. The name of the temporary database table is configurable in the ADIF import scheme. By default, it is named `ADIF_ARGS`.

     > (i) For each parameter, a row is displayed in the database table with two columns:
     >
     >   - `ARG_NAME` for the storage of the parameter name.
     >
     >   - `ARG_VALUE` for the storage of the parameter value.

3) For each parameter that you would like to use in one of the import entries in the scheme, right-click the **Parameters** child node of the ADIF import scheme and select **Create Parameter** from the dropdown list.

4) Click the newly added **@Parameter** node and define the following attributes in the attribute window:

- **Name**: Define a unique name for the parameter. The parameter name must start with @ and must not include other special characters or whitespaces. It can be used in the SQL commands and attributes of the ADIF import scheme and is substituted with the current value of the parameter during execution of the ADIF import scheme.

- **Parameter Type**: Select the data type of the parameter value from the dropdown list.

  > ℹ️ The parameter types StringArray and ReferenceArray can be used if a query shall define that a string or the `REFSTR` of a referenced object shall be within an allowed range of string or `REFSTR` values.

- **Default Value**: Optionally, define a default value that shall be used if no parameter value is provided during execution of the ADIF scheme. Click the **Browse** button on the right of the attribute's field to open the editor for definition of the default value. Note the following about the specification of values:

  - For the definition of dates enter the date into the **Value** field in the format defined in the culture setting of the language you are currently using to render the Alfabet Expand user interface or click the **Select** button and select a date from the calendar.

  - If the parameter is placeholder for a string and used in a query in a `WHERE` condition that checks whether a value is one of a range, the **Parameter Type** must be `StringArray`. The **Value** field in the editor of the **Default Value** attribute displays a text field for `StringArray` parameters. Define the range of allowed values in the text field with each value in a separate row. In the query referring to the array, the parameter must be added in brackets:

    ```
    WHERE app.OBJECTSTATE IN (@appState)
    ```

  - If a string value contains a single quote (like for example in O'Hara), the single quote will be automatically escaped with a second single quote when the parameter value is processed. Escaping the single quote with a second single quote is a requirement for including it into SQL queries. For security reasons, the second single quote is also added in strings to prevent SQL injection.

  - % can be used as wildcard in strings and texts. It is not allowed to define a wildcard in a value of a string array.

  - Boolean values can be selected from a dropdown list. Please note that object class properties of the type `Boolean` that are neither mandatory nor having a default value defined might be set to either true, false or NULL. If the default value is `False`, only object class property values defined as false will be returned, while NULL values will not be returned. This behaviour can be handled via the SQL commands in the ADIF scheme.

  - For ReferenceArray properties no default values can be defined.

- **Mandatory**: Select `True` if the ADIF execution should fail with an error message if no value is defined for the current execution. Select False, if the ADIF execution should be executed even if the parameter is not defined for the current execution. The parameter will then be

substituted with the value defined in the attribute **Default Value**. If no default value is provided, the parameter will be substituted with NULL.

⚠️ In ADIF execution, WHERE clauses in SQL commands containing a parameter are not removed from the query if no value is provided. To make sure that the query result is meaningful if the parameter value is not provided during ADIF execution and the parameter is therefore set to NULL, you should define the query to accept NULL values. For example, to ensure that results are returned if a date compared to an application's start date is not returned, the WHERE clause should be written as:

```
WHERE APPLICATION.STARTDATE >= @StartDate OR @StartDate IS
NULL
```

5) Use the parameters defined in the **Parameters** folder in the ADIF import entries of the ADIF import scheme in the following settings according to demand:

- In the SQL queries defined for any SQL command.

- In the **Message** attribute of SQL commands of the **Result Type** DebugMessage.

- In the **Connection String** attribute of a database import set.

To include a parameter, substitute the actual value with the parameter name within the query or the text string. Please note that parameters are to be included without any single quotes around the parameter name. If the parameter type is requiring definition of single quotes around the value, this is handled by the software during substitution of the parameter name with the parameter value.

👁 For example, a query using the string parameter @orga may be written as:

```
SELECT app.NAME, app.VERSION

FROM APPLICATION app, ORGAUNIT org

WHERE app.RESPONSIBLEORGANIZATION = org.REFSTR

AND org.NAME LIKE @orga;
```

If it is substituted during runtime with the value Trade%, the query will read:

```
SELECT app.NAME, app.VERSION

FROM APPLICATION app, ORGAUNIT org

WHERE app.RESPONSIBLEORGANIZATION = org.REFSTR

AND org.NAME LIKE'Trade%';
```

A debug message defined in the attribute **Message** of an SQL command of the **Result Type** DebugMessage to be written in the import log file during import of the application data can refer to the current organization as well:

```
Import of applications with responsible organization @orga.
```

During runtime, it will be converted to:

```
Import of applications with responsible organization Trade%.
```

StringArray or ReferenceArray parameters need to be included into the query as follows:

```
WHERE APPLICATION.STATUS IN (@StringArray)
```

Defining Parameter Values On Execution of ADIF Schemes

Parameters values are defined during runtime on execution of the ADIF import or export. The parameter name is case sensitive. When defining parameters for execution of the ADIF import or export, you must refer to the parameter name exactly as written in the ADIF scheme, including the starting @. Note the following about the definition of values for the parameters:

- If a string value contains a single quote (like for example in O'Hara), the single quote will be automatically escaped with a second single quote when the parameter value is processed. Escaping the single quote with a second single quote is a requirement for including it into SQL queries. For security reasons, the second single quote is also added in strings to prevent SQL injection.

- Parameter values for data types like string or date are defined without single quotes at the beginning and end. If single quotes are required in the query for the data type, they will be automatically added by the ADIF mechanisms.

- Boolean values must be defined as `1` (true) or `0` (false):

      @MyBooleanValue 1

- String array and ReferenceArray values must be separated with \r\n:

      @MyArrayValue First\r\nSecond

Whether and how command line parameters can be defined at runtime depends on the way the ADIF scheme is executed:

- **Starting ADIF via the console application**

To specify parameters in the command line of the ADIF console application, start the console application with the command line option -<parameter name> <parameter value>. A command line option must be specified for each parameter.

> To specify the parameters `@lastupdate` and `@orga`, the command line could for example be defined as:
>
> ```
> ADIF_Console.exe -import -msalias Alfabet -alfaLoginName
> UserName -alfaLoginPassword pw1234 -scheme AppImport -
> importfile C:\Import\Applications.zip -@lastupdate 09/06/2018
> -@orga ITGroup
> ```

> For general information about starting ADIF via the console application, see Executing ADIF via a Command Line Tool.

- **Starting ADIF via a RESTful service call to the RESTful API of the Alfabet Web Application**

Parameters are defined in the JSON body of the request in the field `UserArgs` returning a JSON object with one field for each parameter in the `format "UserArgs":`
`{" arg1name ":" arg1value "," arg2name ": " arg2value "}`. The field name must be identical to the variable name and the field value defines the variable value for the current execution of the ADIF import scheme.

> To specify the parameters `@lastupdate` and `@orga`, the JSON body of the request could for example be defined as:
>
> ```
> {
> ```

```
                    "Scheme": "AppImport",

                    "UserArgs": {"@lastupdate":"09/06/2018","@orga":"ITGroup"},

                    "Verbose": false,

                    "Synchron": true

                    }
```

> ⓘ If the specification of parameters is not correct in the call, the ADIF job is started, but execution might fail. Information about the success of the ADIF job execution is not returned to the RESTful service call. The return value from the RESTful service call will inform about the last successful execution step, which is start of the ADIF job. The success of the ADIF job execution can then be checked for example via the **ADIF Jobs Administration** functionality.
>
> For more information about the ADIF Jobs Administration functionality, see Executing and Controlling ADIF via the ADIF Jobs Administration and My ADIF Jobs functionalities in the Alfabet User Interface.

- **Starting ADIF via the ADIF Jobs Administration functionality in the Alfabet User Interface**

The ADIF jobs administration functionality does not provide a mechanism to set parameter values. If an ADIF job shall be started via the **ADIF Jobs Administration**, it should not contain parameters. If the ADIF scheme includes parameter definitions, it can only be started via the **ADIF Jobs Administration** functionality with the default values defined in the ADIF scheme for the parameters. The scheme must not include mandatory parameter definitions.

> ⓘ For general information about starting ADIF via the **ADIF Jobs Administration** functionality, see Executing and Controlling ADIF via the ADIF Jobs Administration and My ADIF Jobs functionalities in the Alfabet User Interface.

- **Starting ADIF via an event**

If you start the ADIF job via an event, parameter values can be handed over during execution of the event via a query defined in the event template. In the attributes of the event template, the following attributes must be set:

- **Variable Names**: If the ADIF scheme contain parameter definitions, the names of the parameters must be written as comma separated list into this attribute. The specification must be case sensitive. The values for the parameters must then be provided with the attribute **Values for Variables via Query** / **Values for Variables via Query as Text**.

- **Values for Variables via Query** / **Values for Variables via Query as Text**: The query must return the values for the parameters defined in the attribute **Variable Names** in the same order. The query can be defined either via native SQL or via Alfabet query. Use the **Value for Variables via Query** attribute to define an Alfabet query. Use the **Values for Variables via Query as Text** attribute to define a native SQL query. Alfabet query language parameters can be used in the query. The base object returned via the Alfabet query language parameter BASE depends on the way an event has been triggered. For events triggered via a workflow or wizard, BASE returns the REFSTR of the object the user was working on in the wizard step or workflow step triggering the event. For events triggered via an REST API call event, the BASE object is the same as the one for the event triggering the event.

The query must return a dataset with the column names identical to the parameter names defined with the attribute **Variable Names**.

For example if **Variable Names** is defined as `@AppName,@AppVersion`, the query might be defined as:

```
SELECT REFSTR, NAME AS'@AppName', VERSION AS'@AppVersion'

FROM APPLICATION

WHERE REFSTR = @BASE
```

For general information about starting ADIF via the Alfabet RESTful services, see the reference manual *Alfabet RESTful API*.

- **Starting ADIF via a button in the filter panel of a configured report or in the toolbar of an Alfabet view**

If an ADIF job is started via a button in the filter panel of a configured report or in the toolbar of an object view, the values for the parameter definition in the configured report can be set as follows:

- If the button is defined for an object view or a configured report that is assigned to a base object class via its **Apply to Class** attribute, the information about the base object the object view or configured report is opened for is returned as `@BASE`. If the ADIF scheme has a parameter definition with the **Name** set to `@BASE` and the **Parameter Type** set to `Reference`, this parameter will be substituted with the `REFSTR` of the current object on execution of the ADIF scheme.

- If the button is defined for a configured report that has filter fields, the values set in the filter fields are handed over as parameter values to the ADIF scheme when the ADIF job is started via the button. In the ADIF scheme, the **Name** of the parameter must be identical with the **Name** of the filter field and the **Parameter Type** must be identical to the data type returned by the filter field.

Please note that the filter field name must start with an @. Standard filter fields that are automatically generated by default in reports of the type `Query` based on an Alfabet query are starting with a colon instead of @ and must be adapted manually.

Comparison between filter fields and parameter names is case sensitive.

If the filters of a configured reports are not set when the ADIF job is executed, no value will be provided for the ADIF scheme parameter. Therefore ADIF scheme parameters filled with values from filter fields that are not mandatory in the configured report should not be defined as mandatory in the ADIF scheme. Either a default value should be defined for the parameters or `NULL` value handling should be taken into account in the SQL commands of the ADIF scheme.

For general information about starting ADIF via a button, see Executing ADIF via a Button in the Alfabet User Interface.

Defining Parameter Values During Testing of the ADIF Scheme

When debugging an ADIF scheme with the ADIF Debugger, no command line can be defined. To set parameters for testing, the parameter values for testing can be written into the ADIF scheme.

To specify parameter values for testing in the ADIF scheme:

1) In the ADIF explorer, click the ADIF scheme that you want to specify parameter values for.

2) In the attribute window of the ADIF scheme, set the following attributes:

- **Debug Arguments**: Define the values for the parameters used in a comma-separated string. The format of the string must be <parameter name>=<value>. Please note that the values must not be written into single quotes. If single quotes are required in the query for the parameter data type, for example for strings, these will be set automatically when substituting the value in the query.

Mandatory parameters must be defined in the **Debug Arguments** to test the ADIF scheme via the ADIF De-bugger. If the parameter is not mandatory and no value is defined for the parameter in the **Debug Arguments**, the ADIF Debugger will use the default value defined for the parameter. If no default value is defined, the parameter will be substituted with `NULL`.

For example, to specify import of applications for which the organization responsible for the application is variable and the last update of application data is variable, the **Debug Arguments** attribute can be specified as:

```
@lastupdate=09/10/2012,@orga=ITGroup
```

Configuring Logging Parameters

During import via the ADIF console application log messages are written to:

- A temporary database table in the Alfabet database.

- The log file of the ADIF console application. By default, this is the log file `ADIF_Console.log` in the working directory of the ADIF console application.

When the ADIF console application is run asynchronously in remote mode, no import-re-lated information will be written to the log file of the ADIF console application. The console application handles the data to the Alfabet Server and closes the process after suc-cessful data submission.

Logging information is written to a log file in the following format:

```
<date and time> <message type> <message text>
```

For example:

```
2010-11-30T10:43:24.31Z DEBUG_INFO Query returns 314 records
```

Log messages are written to the log file in the language defined in the cultures of the Alfabet database. When the ADIF console application is run with an server alias, the culture setting defined as the default in the configuration of Alfabet is used. When the ADIF console application is run with a remote alias, the cul-ture setting of the remote alias configuration is used.

For information about configuring cultures for the Alfabet database, see the section *Specifying the Cultures Relevant to Your Enterprise* in the chapter *Localization and Multi-Language Support for the Alfabet Interface*. in the reference manual *Configuring Alfabet with Alfabet Expand*.

For information about specifying the culture setting of the remote alias, see the section *Configu-ration Attributes for the Alfabet Components* in the reference manual *System Administration*.

The timestamp is the UTC time (coordinated Universal Time) and may therefore differ from the time in your local time zone. The timestamp is written in ISO 8601 combined date and time format as year-month-day-Thour:minutes:secondsZ.

The message type can be one of the following:

- **ERROR**: An error occurred. The message describes the type of error.

- **WARNING**: Problems were encountered during the execution of the utility that are not as severe as an error. The process was executed but the result should be checked. The message describes the problem.

- **INFO**: Information about the normal execution of the utility is given.

- **DEBUG_INFO**: Information about the import process is given. This information is also stored in the Alfabet database in a temporary database table.

> ⓘ The debug info returns the number of records that are changed during the import process. This number can be higher than the number of objects directly affected by the import process. For example, if the ADIF scheme triggers deletion of objects from the Alfabet database, objects that are defined as dependent objects are also deleted or changed. For example, references to the deleted objects have to be removed from properties of other objects or from the `RELATIONS` table. The **DEBUG_INFO** returns the overall number of objects changed, created or deleted in the database, which may be multiple objects per deleted object.

During the import process, emails containing log information can be sent in a defined time interval to a configurable email address.

> ⓘ The email is meant as an alive message of the system informing the recipient about the availability of an active ADIF process. The content of the email is limited to the one-line content that is written to the log file at the moment the email is generated and is therefore not suitable for debugging.

The storage of log information and the sending of emails with process-related information are configurable either in the ADIF import scheme configuration or in the command line when starting the ADIF console application. When a configuration can be done both in the ADIF import scheme and the command line, the specification via the command line options overwrites the configuration in the ADIF import scheme.

The following table gives an overview of the configuration options:

Configurable Logging Behavior	Attribute of the ADIF Scheme	Command line Option	Description
Time interval for sending emails containing current log information	Debug Heart Beat	`-heartbeat <time in minutes>`	Specify the time interval in minutes between sending emails with the current log message of the ADIF console application during import. By default, the debug heart beat is set to -1, that means no emails are sent. **NOTE**: If the import process is finished before the time interval elapsed for the first time, no email is sent.

Configurable Logging Behavior	Attribute of the ADIF Scheme	Command line Option	Description
email address of the recipient of the log information	Recipient Mail	`-recipient-mail <email address>`	Specify the email address of the person that shall receive the current log message of the ADIF console application via email in the configured time interval.
email address used as sender address in the emails with log information	Sender Mail	`-sendermail <email address>`	Specify the email address that shall be used as sender email address for emails sent via the ADIF console application.
Name of the log file of the ADIF console application	-	`-logfile <filename>`	Specify the name of the log file of the ADIF console application. Allowed file extensions are.log and.txt. The default name is `ADIF_Console.log`
Location of the log file of the ADIF console application	-	`-logpath <path>`	Specify the path to the log file of the ADIF console application. By default the log file is stored in the working directory of the ADIF console application. The ADIF console application process must have write access rights to the specified directory.
Amount of data written to the log file	-	`-logverbose`	If set, all information about the process is written to the log file. If not set, only error messages and information about the process start and end is written to the log file.
Removal of old log messages from the log file	-	`-logclear <number of days>`	If `-nologappend` is set, a new log file is created each time the utility is used with the same specification of `-logfile` and `-logpath`. The log file name is extended with a timestamp specifying the current UTC time. If `-nologappend` is not set, logging information is appended to the existing log file each time the utility is used. **NOTE** : To restrict the file size, you can set the `-logclear` option to delete old log messages.
Creation of a new log file for each process started	-	`-nologappend`	This option can only be used if `-nologappend` is not set. If `-nologappend` is set, the `-logclear` setting is ignored. During logging, the log file is scanned for log messages with a timestamp older than the number of

Configurable Log-ging Behavior	Attribute of the ADIF Scheme	Command line Option	Description
			days specified with `-logclear` and these messages are deleted.

NOTE: The scanning process can lead to drawbacks in performance. |

Configuring Log Message Content

ADIF export schemes can be configured to include customer-defined log messages and data sets found via SQL commands in the log files during execution of the ADIF import scheme. Log file content can be added to the ADIF import scheme via SQL command elements. SQL command elements for debug messages can be located in any **SQL Commands - <Action Type>** folder except the **SQL Commands - DataBind** folder.

During import execution, the SQL commands in the ADIF import scheme are executed in the order specified by the command location in the XML definition. When the SQL command is due for execution, the message defined in the SQL command element and, if specified, the result of the SQL query as data set, are displayed in the log message.

In an ADIF import scheme that imports data about devices from an XML file, information about the number of devices that are updated in the Alfabet database during import shall be written to the log file during the import process. An SQL command of the **Result TypeDebugDataSet** is added to the **SQL Commands - OnComplete** folder of the import entry that triggers the import of device data from the XML file:

In the property window, the attribute **Result Type** is set to `DebugDataSet`. The data set to be displayed in the log file is defined by the SQL query defined in the attribute **Text**. The SQL query defined in the example finds the name and version of the devices in the temporary table that were bound to an existing object in the Alfabet database.

Prior to the data set, a text explaining the data set shall be displayed in the log file. Therefore, an explanatory text will be added in the **Message** attribute.

When executing the import, the **Message** text and the data set resulting from SQL query execution will be added to the log file when the SQL command is due to be executed in the processing order of the ADIF import scheme:

```
2011-04-26T15:41:08.333Z DEBUG_INFO  The following devices are updated by the import process:
NAME|VERSION
Linux eBank Server|1
Windows eBank App-Server A|1
Windows eBank App-Server B|1
```

(i) Debug messages and data sets can refer to the current import conditions via variables that de-
pend on command line arguments of the `ADIF_Console.exe` when starting the import.

For information on the use of variables to refer to the conditions defined in the command line,
see *Configuring Execution of the Import Scheme Dependent on Current Parameters*.

Inserting a Static Text Message

To add a static text message to the log file:

1) In the explorer of the ADIF configuration interface, right-click the folder **SQL Commands - <Action
Type>** that is executed in the order of commands when you want the text message to be
displayed.

2) In the context menu, select **Create SQL Command**. A new SQL command 🗲 is added as a child
node of the **SQL Commands - <Action Type>** folder.

3) Click the new SQL command 🗲 and specify the following attributes in the attribute window:

- **Name**: Enter a meaningful name for the SQL command.

- **Comments**: Enter a comment that provides information about the functionality implemented
by the SQL command.

- **Result Type** : Select `DebugMessage`.

- **Message**: Enter a text that shall be written to the log file at the position of the execution of
this SQL command.

- **Is Active** : Select `True` to activate the SQL command. Select `False` to deactivate the SQL
command.

Inserting a Data Set

To include data from the database in the log file:

1) In the explorer of the ADIF configuration interface, right-click the folder **SQL Commands - <Action
Type>** that is executed in the order of commands when you want the text message to be
displayed.

2) In the context menu, select **Create SQL Command**. A new SQL command 🗲 is added as a child
node of the **SQL Commands - <Action Type>** folder.

3) Click the new SQL command 🗲 and specify the following in the attribute window:

- **Name**: Enter a meaningful name for the SQL command.

- **Comments**: Enter a comment that provides information about the functionality implemented by the SQL command.

- **Result Type** : Select `DebugDataSet`.

- **Message**: Enter a text that shall be written to the log file at the position of the execution of this SQL command prior to the data set resulting from the SQL query defined.

- **Text** : Define an SQL query with a `SELECT` statement that returns a data set. The resulting data set is written to the log file at the position that indicates the execution of the SQL command. If a text is also defined in the **Message** attribute, this text is written to the log file prior to the data set. You can either write the SQL query directly into the attribute field or click the **Browse** [...] button to open an editor for the definition of the SQL query. Define the SQL query in the **SQL Text** tab of the editor.

- **Message**: Enter a text that shall be written to the log file as an introduction to the data set.

- **Ignore Errors** : Select `True` if you want the export to be executed even if the SQL statement specified with the attribute **Text** results in an exception. Select `False` if you do not want the import to be executed in case of an error in the SQL statement.

- **Is Active** : Select `True` to activate the SQL command. Select `False` to deactivate the SQL command.

To set the parameters in the ADIF import scheme:

1) In the explorer of the ADIF configuration interface, click the import scheme that you want to configure. The attributes of the ADIF import scheme are displayed in the attribute window on the right.

2) In the attribute window, set the parameters listed in the table above as applicable.

Configuring the Import Audit History

In Alfabet, object classes can be configured to have an audit history.

For information about how audit information is stored in the database, see Audit History Storage in the chapter The Alfabet Meta-Model in the Alfabet Database.

The ADIF import scheme allows you to configure the audit history storage during the ADIF import.

⚠ The ADIF import definitions should never include the audit tables. Changing the audit table for an object can lead to severe database inconsistencies and may result in an error occurring.

Clean-up of the audit tables

Sometimes a change is performed to an object which overwrites the value for a property with the value already available in the database table. For example, the property State of an object is set to value 'Plan' even though the value was already 'Plan'. This is in fact not a change from the logical perspective, but technically a redundant `UPDATE` statement is executed. When such a change is performed to the object, a new line is added to the audit table. This line indicates that a change has been made, but the entry is obsolete in the audit table because all object class properties remain unchanged.

The ADIF interface allows the import scheme to be configured so that the obsolete lines are automatically deleted/consolidated in the audit table. Configuration is done per **Import Entry**, because the setting is only required for classes for which an audit table exists.

> ⚠️ It is recommended to configure audit table clean-up for all object classes that are configured to have an audit history. Otherwise, users viewing the audit history via the Alfabet interface will see useless information and the size of the Alfabet database will increase considerably.

1) In the explorer, click the **Import Entry** that you want to configure auditing for.

2) In the attribute window, set the **Squeeze Audit Table** attribute to `True`.

Information about the Import User

The user who is triggering the ADIF import is defined as the user performing the change. When starting the ADIF console application, the user specified via the command line option `-alfaLoginName` is added as user performing the changes caused by the import process into the audit tables. If the command line option `-alfaLoginName` is not used, `ALFABET_INTERN` is written into the audit tables to specify the user responsible for the changes.

> ℹ️ The current user name is also used to set properties of objects that are created or changed during import. All properties in the group `Artifact` have properties that store information about changes to the object.
>
> The following properties are set when an object is created or updated:
>
> • The property `LAST_UPDATE` is set to the current date.
>
> • The property `LAST_UPDATE_USER` is set to the current user.
>
> The following properties are additionally set when an object is created:
>
> • The property `CREATION_DATE` is set to the current date.
>
> • The property `CREATION_USER` is set to the current user.
>
> During import, the `LAST_UPDATE_USER` and the `CREATION_USER` are specified as `ALFABET_INTERN` if the command line option `-alfaLoginName` is not set.

Configuring the Automatic Start of Workflows During Import

Alfabet supports your enterprise in defining and maintaining workflows in which you can track and coordinate the activities that should be performed by various persons in a particular sequence.

> ℹ️ For more information about the workflow functionality, see *Executing Workflows and Participating in Workflow Steps* in the reference manual *Getting Started with Alfabet*.

Workflows can be automatically started as part of the import triggered by ADIF. This may be useful to ensure that new data in the Alfabet database is post-processed by users responsible for data maintenance.

The automatic start of workflows requires the following configuration:

- A workflow template must be defined in Alfabet Expand.

- An element **Workflow Entry** must be added to the ADIF import scheme that specifies the name of the workflow template.

The definition of workflow templates is described in the chapter *Configuring Workflows* in the reference manual *Configuring Alfabet with Alfabet Expand*.

Take the following into account when configuring the workflow template and the ADIF import scheme:

- The workflow must be configured to be started automatically.

- Make sure that the imported objects can be identified via a query in the workflow template to enable starting of workflows for imported objects only. One method is the definition of a custom property for the target class for import in the Alfabet meta-model using Alfabet Expand. The custom property is then filled during import e.g. with the import date that is filled during import.

For details, see *Use Case: Defining Import-Related Custom Properties* in the section *Configuring SQL Commands for Optional Enhanced Import Features*.

For example, a new custom property `ImportDate` (Techname: `IMPORTDATE`) is defined for a target object class, for example for the object class `Device`. The `ImportDate` property has the data type `Date` and is filled with the current date during update. The workflow template can now be configured with the following query to find the objects for which workflows will be started:

```
SELECT REFSTR

FROM DEVICE dev

WHERE dev.IMPORTDATE = @TODAY
```

The query finds all objects of the object class `Device` for which data was updated or created on the current day. As the query is executed during import, the import date matches the start date of the workflow.

- Start of workflows is executed during import after execution of all import entries, but prior to dropping temporary tables. Nevertheless, temporary tables are ignored by Alfabet functionalities and the queries defined for workflows do not interpret data from temporary tables.

- Workflow entries are ignored when the attribute **Commit After Run** of the import scheme is set to `False`.

To add a workflow entry to the ADIF import scheme:

1) In the explorer of the ADIF configuration interface, right-click the ADIF import scheme and select **Create Workflow Entry**. A new node is added to the explorer with a default name for the workflow entry. Under the workflow entry node, a folder for the creation of SQL commands to be executed on activation of the workflow entry node is automatically added.

2) In the attribute window of the workflow entry, set the following attributes:

- **Name**: Enter a meaningful name for the workflow entry.

- **Comments**: Enter a comment that provides information about the functionality implemented within the workflow entry.

- **Workflow Template**: In the drop-down list, select the name of the workflow template.

- **Is Active** : Select `True` to activate the execution of all import entries within the import set. Select `False` to deactivate the execution.

Chapter 5:   Configuring Data Export with ADIF

This chapter guides you through the process of data export from the Alfabet database with the ADIF export scheme.

The Export Process

Data can be exported from Alfabet database tables to database tables of external databases, Microsoft® Excel® files, comma-separated data format files (.csv or.txt) and XML files.

Export is triggered by ADIF export schemes that are configured by the customer via SQL commands. An export scheme can trigger export to multiple database tables or files of one or different file formats.

⚠️ If you export of a huge amount of data to a single file, the time required for processing the data and the memory usage during processing of data may be exceedingly high. To avoid performance problems it is recommended to configure the ADIF export scheme to export data into multiple files. this can for example be performed to filter object export on basis of a value of one of the object class properties. For example, by exporting all applications with a name starting with A into one file, all starting with B into another file and so on.

The export process is started via the ADIF debugger or via the ADIF console application.

When starting the ADIF export, the following process is executed:

1)   Validate the export scheme. Export is stopped in case validation fails.

2)   Create temporary directory on server side to use it for log and export files.

3)   Write command line arguments to the Alfabet database. If the export is triggered by an export scheme located in an XML file, the export scheme is also written to the database.

4)   Perform export steps configured in the ADIF export scheme. The following descriptions of export configuration provide information about which specific action is triggered by the individual elements of the scheme. During export execution, export files are created in the temporary directory for export to XML, CSV or Microsoft® Excel® formats. For export to external database tables, data is written to the external database.

5)   Create ZIP file from all files generated in the temporary directory.

6)   Upload the ZIP file to the **ADIF_SYS** folder of the **Document Explorer** of the Alfabet database.

7)   Upload log information about the export process to the **ADIF_SYS** folder of the **Document Explorer** of the Alfabet database.

8)   Delete the temporary directory on the server-side created prior to export execution and all content of the directory.

9)   Download ZIP file to the location specified in the command line arguments.

10)   Download the log file content to the console log file.

11)   Unzip ZIP file if the unzip action is requested with the command line parameters.

This chapter describes how to configure different export features within the ADIF export scheme. The export process is highly configurable to meet individual customer requirements concerning the resulting data output.

Basic Configuration of Data Export

An ADIF export scheme is an XML object that can be edited either in an XML editor or in the ADIF configuration interface that is provided as part of the configuration tool Alfabet Expand. In the ADIF configuration interface, the XML elements are displayed as nodes in the explorer, while the attributes of an XML element are displayed in the attribute window on the right.

All information and commands required to export data from the Alfabet database are defined via the XML elements of the ADIF export scheme. The XML elements must follow a defined sequence that is determined by an XSD scheme. In the ADIF explorer, the generation of elements via the context menu of the explorer nodes guides you through the creation of all required elements. Only elements that are allowed as sub-elements can be created as child nodes of an explorer node. When you define the ADIF export scheme in an XML editor, you must consult the XSD scheme for information about the permissible sequence of objects.

The following elements are part of the ADIF export scheme:

XML Element Name	Caption in the ADIF Interface	Purpose
`ADIF_ExportScheme`	Export Scheme	Element for the configuration of basic overall export execution parameters.
`XmlExportSet`	Entry for Classes (XML)	Structuring element that is a container for elements defining the export to XML files.
`FileExportSet`	Entry for Classes (Excel)	Structuring element that is a container for elements defining the export to Microsoft® Excel® files or CSV files. **NOTE**: Via the ADIF interface this element is only available for the structuring of Excel Export Entries.
`DBExportSet`	DB Export Set	Element defining the connection to an external database.
`ExportEntry`	Entry	Element defining the content of the export to one external database table or one external file.
`SQLCommand`	SQL Command	Sub-element of `ExportEntry` elements for the definition of SQL Commands to be executed during export.
`AlfaMethod`		Sub-element of `ExportEntry` elements to assign custom code to the entry. AlfaMethod elements are only relevant if custom code has been developed by Software AG for special customer requirements. Information about the configuration is provided individually with the custom code.

This section describes the basic configuration required to trigger simple data export to different file formats or to external database tables. The basic configuration can be enhanced to allow complex export scenarios by adding the special features described in the following sections:

-

-

    -

    -

-

    -

    -

Creating an ADIF Export Scheme

Do the following to create a new ADIF export scheme:

1) In the explorer, right-click the **ADIF Schemes** root node and select **Create Export Scheme**. The new export scheme is added to the explorer and the attribute window of the new export scheme is displayed on the right.

2) In the attribute window, set the following attributes for the ADIF export scheme:

    - **Name**: Enter a unique name. The **Name** is used to identify the ADIF scheme in technical processes. It must be unique and should not contain whitespaces or special characters.

    - **Caption**: Enter a meaningful and unique caption. The caption is used to identify the ADIF import scheme in the Alfabet user interface in the **ADIF Jobs Administration** and **My ADIF Jobs** functionalities.

    - **Description**: Enter a meaningful and short description of the result of the ADIF import. The description is displayed in the Alfabet user interface in the **ADIF Jobs Administration** and **My ADIF Jobs** functionalities.

    - **Commit After Run** : If set to `True`, the result of the data export is written persistently to the external files or the external target database. If set to `False`, the export process is rolled back after execution and no changes are written to the external database or files. It is recommended to set **Commit After Run** to `False` for a new export scheme for export to an external database to allow debugging without the risk to corrupt the external database. After successful testing of the data export and verification that the resulting changes to the external database are as expected, **Commit After Run** can be set to `True` to perform regular data export.

        ⚠ The following restrictions apply to setting **Commit After Run** to `False`:

        - **Commit After Run** only affects database transactions. If you export data to a file, the export file is created, and data is added to the file also if **Commit After Run** is set to `False`.

        - Setting **Commit After Run** to `False` rolls back all changes cause by DML statements (changes to data records in existing tables). Creating or deleting

tables is not included in the roll back. That means, for example, if you test an ADIF scheme that is configured to write temporary tables to the database persistently, these temporary tables will be created persistently even if **Commit After Run** is set to `False`. SQL Commands of the type `OnActivate` are also excluded from roll back.

- **Alfabet User Interface Behaviour**: Select one of the following to define the availability of the ADIF scheme in the Alfabet user interface:

    - `VisibleExecutable`: The ADIF export can be triggered in the **ADIF Job Administration** functionality and the success of the ADIF jobs executed with this ADIF scheme can be controlled via the **ADIF Job Administration** and **My ADIF Jobs** functionalities.

    - `VisibleNotExecutable`: The success of the ADIF jobs executed with this ADIF scheme can be controlled via the **ADIF Job Administration** and **My ADIF Jobs** functionalities, but the ADIF export cannot be triggered via the Alfabet user interface.

    - `NotVisible`: The ADIF scheme and the information about ADIF jobs executed with this ADIF scheme are not visible in the **ADIF Job Administration** and **My ADIF Jobs** functionalities. This is the default value for new ADIF export schemes.

    > ℹ️ For information about administration and execution of the ADIF scheme via the **ADIF Job Administration** and **My ADIF Jobs** functionalities, see Executing and Controlling ADIF via the ADIF Jobs Administration and My ADIF Jobs functionalities in the Alfabet User Interface.

- **Applicable for REST API**: Set the attribute to **True** if the ADIF scheme shall be executed via a RESTful service call to the endpoint `adifexport` of the Alfabet RESTful API either via a RESTful service call from an external RESTful client or via an Alfabet event that is triggering the execution of the ADIF scheme when a user enters or leaves a wizard or workflow step or when an event for execution of a RESTful service call to a third party application is finished.

    > ℹ️ Execution of the ADIF scheme via the Alfabet RESTful API requires setup of the RESTful services. For information about the requirements and the execution of the service call, see the reference manual *Alfabet RESTful API*.
    >
    > For the additional configuration to implement execution of an event, see *Configuring Events*.

After creating the export scheme, you can now add export definitions to trigger export to one or multiple data formats. Proceed with:

- Configuring Export to External Database Tables

- Configuring Export to XML Files

- Configuring Export to Comma-Separated Data Files

- Configuring Export to Microsoft® Excel® Files

Optionally, you can configure the following options:

- Configuring Execution of the Import Scheme Dependent on Current Parameters

- Configuring Logging Parameters

Configuring Export to External Database Tables

Data can be exported to database tables of external databases.

Data export is triggered by an SQL `SELECT` statement. The result of the SQL query is a data table that is included in the external database as follows:

- Data is written to the database table specified with the attribute **Export Table** of the export entry for which the SQL query is defined.

- Data is mapped to the columns in the specified table by mapping the alias names of the `SELECT` statement to the rows of the external database table.

- All records from the data set are added to already existent records of the external database table.

> ⚠️ If data is added to a database table that already contains data, a simple data export will not be able to update the existing data but will only add new rows to the database table. If this leads to unique key violations (because another object with the same key already exists in the database table), the data export will be aborted.
>
> The ADIF export scheme offers various mechanisms to solve this problem:
>
> - Additional SQL commands can be configured to manipulate the target database prior to or after data export. You can configure the export process to write the exported data to a temporary table, update the actual target database table via appropriate SQL commands, and then drop the temporary table.
>
> - A mechanism can be activated that deletes all records from the target database table prior to the execution of data export.
>
> For more information about the update of existing data in a target database, see Use Case: Filling Empty Property Values with Default Values When Importing to Mandatory Object Class Properties in the section Configuring Logging Parameters.

> ⚠️ If a string is imported into a target database exceeds the maximum allowed size for strings in the target database, the data export will be aborted. Therefore, if string size limitations apply to the values in the target database, the SQL command defined for data export must be defined to truncate data to the allowed file size. The native SQL functions `LEFT(Column,MaximumLength)` or `SUBSTRING(Column,1,MaximumLength)` can be used to truncate data strings.

Configuring Export to a Target Database

The following configuration within an ADIF export scheme is required for data export from the Alfabet database to an external database:

- Create an export set for the specification of the connection of the external database.

- Create an entry for the specification of the target database table.

- Define the SQL query for the definition of the content that shall be added to the external database table in the sub-folder **SQL Commands - DataExport** of the export entry.

⚠ You are not allowed to define more than one SQL command of the type `DataExport` for export to a target database.

The workflow for the definition of the export entry is described below.

Optionally, you can do the following:

- Define additional SQL commands to define complex export scenarios like, for example, the update of data in a database table instead of simple addition of rows to the existing database table. For more information, see Configuring Logging Parameters.

To create an export entry for export to an external database:

1) In the explorer of the ADIF configuration interface, right-click the ADIF export scheme to which you want to add an export set and select **Create DB Export Set**. A new database export set is added to the explorer.

2) Select the new database export set in the explorer, and edit the following in the attribute window:

- **Name**: Enter a meaningful name for the database export set.

- **Comments**: Enter a comment that provides information about the functionality implemented within the database export set.

- **Driver Type**: Select the type of database server that the external database is located on. Import can be performed from databases on Microsoft® SQL Servers®, Oracle® Database servers or Microsoft® Access databases. Select one of the following:

  - `SqlServer` for access to Microsoft SQL Server

  - `Oracle` for access to Oracle

  - `Access` for access to Microsoft Access via an OLE driver

  - `ODBC` for access to Microsoft Access, Oracle® MySQL or PostgreSQL 9.1 via an ODBC driver

  - `Hadoop` for access to Hadoop systems

- **Driver Sub-Type** : This attribute is only visible if `SqlServer` is selected in the **Driver Type** attribute. Select the driver that shall be used for the connection to the Microsoft SQL Server:

  - `MSNetServer`: The driver used in previous Alfabet releases. It is restricted to use with.NET Framework. This is the default value.

  - `MSSqlServer`: A driver supporting both.NET Framework and.NET Core environments.

- **Connection String**: Enter the connection string required to connect to the external source. The connection string depends on the database server used. It typically contains source location, user name, and other parameters, according to the requirements of the database server.

⚠ It is recommended that you define the database login in the connection string via a user name and password. If Windows Authentication is used for login and the ADIF Console Application is started with a remote alias, the connection to the database will only be successful if the Alfabet Server is started with the same domain user name and access rights that are provided with the connection string.

ⓘ Server variables can be used in the connection string. Server variables allow you to define all or part of the definition in the server alias configuration of the Alfabet Server instead of directly defining it in the ADIF scheme. Use of server variables is useful, for example, when using the configuration in a test and production environment with different external sources. The same external source definition can be used in both environments. The server alias definition used in the test and production environment define the correct connection data for the external source used in the respective environment. The use of server variables is described in the section *Using Server Variables in Web Link and Database Server-Related Specifications* in the chapter *Configuring Reports*.

ⓘ ODBC can be used for connections to Microsoft Access or PostgreSQL 9.1 database servers, but performance will be lower compared to native drivers and stored procedures are not supported.

◎ The following are examples of connection strings for connections to different database servers.

Connection string for connection to a Microsoft SQL Server® with standard login:

```
Server=myServerAddress;Database=myDataBase;User
Id=myUsername;Password=myPassword;
```

Connection string for direct connection to an Oracle® database server:

```
User Id=APP_ID;Password=*********;Direct=true;Data
Source=servername;Port=Port;SID=orcl
```

Connection string for direct connection to an Oracle® databse server including the service name:

```
User
Id=********;Password=*********;Server=ServerFQDN/IP;Unicode
=True;Connection Timeout=60;Max Pool
Size=250;Direct=True;Service Name=ServiceName;Port=1521
```

Connection string for indirect connection to an Oracle® database server:

```
User
Id=***;Password=*****;Server=orcl;Unicode=True;Connection
Timeout=60;Max Pool Size=250
```

- **Is Active** : Select `True` to activate the execution of all export entries within the export set. Select `False` to deactivate the execution.

3) Right-click the database export set and select **Create Entry**. A new export entry node is added to the explorer as sub-node of the database export set.

4) In the explorer, click the new export entry node.

5) In the attribute window on the right, set the following attributes for the export entry:

- **Name**: Change the default name of the entry to a meaningful name. The **Name** is used to identify the export entry in technical processes. It must be unique and should not contain whitespaces or special characters.

- **Comments**: Optionally enter a comment that provides information about the export for internal use.

- **IsActive** : Select `True` to activate the execution of the entry during export.

- **Export Table**: Enter a name for the target database table in the external database. The data will be written to the specified table.

- **Delete Table After Export**: Select `True` to delete the database table specified as target for the export with the attribute **Export Table** after execution of the export. The table is deleted after all SQL commands specified for the export entries in the database export set were executed. Select `False` to write data to the target database persistently.

  ⓘ When you want to update existing data in a target database, you can export to a temporary table, specify additional SQL commands to be executed on the target database to merge the data to the actual target database table, and delete the temporary table after export. The creation of the temporary table must be triggered with an SQL command, while dropping of the table is done automatically when the **Delete Table After Export** attribute is set to `True`.

- **Empty Table Before Export**: Select `True` to delete all records from the database table specified as target for the export with the attribute **Export Table** prior to execution of the export. The table is deleted after all SQL commands specified for the export entries in the database export set. Select `False` to add data to existing data records.

  ⓘ Unique key and primary key settings depend on the configuration of the target database and can therefore not be taken into account by the standard export mechanism provided via ADIF. Therefore, the addition of data to exiting records can lead to problems during data import. Deleting records from a table before data export is executed is part of the mechanisms that help to implement an export configuration that allows to update existing data to a target database. For more information about such configuration, see Use Case: Filling Empty Property Values with Default Values When Importing to Mandatory Object Class Properties in the section Configuring Logging Parameters.

- **Boolean'False'** : If a boolean property exported to an external database shall be represented by a string in the target database, select the string value that shall be used if an attribute of the data type Boolean is set to `False`. The default is `True`.

  ⓘ When a boolean property is not mandatory in the Alfabet meta-model and therefore set to NULL for some objects, NULL is not interpreted as `False` but as an empty data field. If you want NULL to be treated as `False`, you must configure your SQL query accordingly.

- **Boolean'True'** : If a boolean property exported to an external database shall be represented by a string in the target database, select the string value that shall be used if an attribute of the data type Boolean is set to `True`. Default is `True`.

- **Number Decimal Digits**: If numbers shall be converted to strings during export, define the number of digits to be defined in the output for the decimal part of numbers. The default is 2. For example, two and a half will be written as 2.50.

- **Number Decimal Separator**: If numbers shall be converted to strings during export, define the decimal separator for numbers. The default is a period.

- **Number Group Separator**: If numbers shall be converted to strings during export, define the separator for number groups. The default is a comma. For example, three thousand will be written as 3,000.

- **Date Format**: If dates shall be converted to strings during export, define the format for dates written to the target database table. The default is dd/MM/yyyy.

6) In the explorer, expand the export entry node of the new database export entry.

7) Right-click the folder **SQL Commands - Data Export** and select **Create SQL Command**. A new SQL command node is added to the explorer.

8) Click the new SQL command node and specify the following in the attribute window:

- **Name**: Enter a meaningful name for the SQL command.

- **Comments**: Enter a comment that provides information about the functionality implemented by the SQL command.

- **Command Type** : Select `DMLstatement` in the drop-down list.

- **Apply To** : Select `local`.

- **Result Type** : Select `Undefined`.

- **Text** : Define an SQL query with a `SELECT` statement that returns a data set. The resulting data set is exported to the target database table specified in the export entry. The column headers resulting from the `SELECT` statement must match the name of the column headers in the target database table. Either write the SQL query directly in the field or click the
  Browse � button to open an editor for the definition of the SQL query. Define the SQL query in the **SQL Text** tab of the editor.

  (i) If you want to configure complex export scenarios like, for example, the update of existing data in the target database, you can add additional SQL commands to the export entry. For more information about the definition of SQL commands within an export entry, see Configuring Logging Parameters.

- **Ignore Errors** : Select `True` if you want the export to be executed even if the SQL statement specified in the attribute **Text** results in an exception. Select `False` if you do not want the export to be executed in case of an error in the SQL statement.

- **Is Active**: Select `True` to activate the SQL command. Select `False` to deactivate the SQL command.

9) Repeat step 3 - 8 for each database table of the target database you want to export data to.

Configuring Export to XML Files

Data can be exported to XML file formats with the extension.xml.

Data export is triggered by multiple SQL `SELECT` statement. The SQL statements can be hierarchically structured to build the hierarchy of XML elements within the target file. This section first describes the structure that must be provided as a framework for the definition of an XML export file in the ADIF export scheme in the section. The definition of the XML in the target file via SQL queries and attributes of the ADIF export scheme is then explained based on a simple example in the section.

Configuring Export to a Target XML File

The following configuration within an ADIF export scheme is required for data export from the Alfabet database to a single XML file:

- Create an export entry for the specification of the target export file.

- Define the SQL query for the definition of the content of the file in the sub-folder **SQL Commands - DataExport** of the export entry.

The workflow for the definition of the export entry is described below.

Optionally, you can do the following:

- Define additional SQL commands to define complex export scenarios. For more information, see Configuring Logging Parameters.

- Structure the export entries in an export set. Export sets do not have technically relevant attributes and are only used to enhance the readability of the ADIF export set. The definition of XML export set is currently only possible via direct definition in the text editor of the `XMLExportSet` element in the XML of the ADIF export scheme. For information about defining ADIF export schemes in text editors, see *Configuring ADIF via XML Editing*.

To create an export entry for export to XML files:

1) In the explorer of the ADIF configuration interface, right-click the export scheme and select **Create Entry**. A new export entry node is added to the explorer as sub-node of the ADIF export scheme.

    Alternatively, you can select the option **Create XML Entry for Classes** to create an XML export entry that already contains a basic configuration of an XML structure with one XML element per class in parallel below the root node.

    If you select **Create XML Entry for Classes**, a class selector opens that allows you to select the object classes for which data shall be added to the first level of XML elements in the XML output. For each class that you select in the class selector, an SQL command generating XML elements for the data output will be automatically added to the XML entry and default values will be set for all attributes. Nevertheless, you should control the output following the next steps of this workflow.

2) In the explorer, click the new export entry node.

3) In the attribute window on the right, set the following attributes for the export entry:

    - **Name**: Change the default name of the entry to a meaningful name. The **Name** is used to identify the export entry in technical processes. It must be unique and should not contain whitespaces or special characters.

    - **Comments**: Optionally, enter a comment that provides information about the export for internal use.

    - **Is Active**: Select `True` to activate the execution of the entry during export.

    - **Export Table**: Enter a name for the target file for the export. The export file will be created during export. It must have the extension.xml

    - **XML Root Tag**: Enter the name of the root element of the XML output of the export.

- **Encoding**: Select the encoding for the data in the export file from the drop-down list.

- **Boolean'False'** : Select the string value that shall be written to the export file when an attribute of the data type Boolean is set to `False`. The default is `False`.

  > (i) When a boolean property is not mandatory in the Alfabet meta-model and therefore set to NULL for some objects, NULL is not interpreted as `False` but as an empty data field. If you want NULL to be treated as `False`, you must configure your SQL query accordingly.

- **Boolean'True'** : Select the string value that shall be written into the export file when an attribute of the data type Boolean is set to `True`. Default is `True`.

- **Number Decimal Digits**: Define the number of digits to be defined in the output for the decimal part of numbers. The default is 2. For example, two and a half will be written as 2.5.

- **Number Decimal Separator**: Define the decimal separator for numbers. The default is a period.

- **Number Group Separator**: Define the separator for number groups. The default is a comma. For example, three thousand will be written as 3,000.

- **Date Format**: Define the format for dates written to the export file. The default is dd/MM/yyyy.

4) In the explorer, expand the export entry node of the new export entry.

5) Right-click the folder **SQL Commands - Data Export** and select **Create SQL Command**. A new SQL command node is added to the explorer.

6) Click the new SQL command node and specify the following in the attribute window:

- **Name**: Enter a meaningful name for the SQL command.

- **Comments**: Enter a comment that gives information about the functionality implemented by the SQL command.

- **Command Type** : Select `DMLStatement` in the drop-down list.

- **Result Type** : Select `Undefined`.

- **Ignore Errors** : Select `True` if you want the export to be executed even if the SQL statement specified with the attribute **Text** results in an exception. Select `False` if you do not want the export to be executed in case of an error in the SQL statement.

- **Is Active** : Select `True` to activate the SQL command. Select `False` to deactivate the SQL command.

- **Text** : Define an SQL query with a `SELECT` statement that returns a data set. The resulting data set is exported to the target file specified in the export entry. Either write the SQL query directly in the attribute field or click the **Browse** button to open an editor for the definition of the SQL query. Define the SQL query in the **SQL Text** tab of the editor.

Definition of the XML Content of the Export File

The content of the XML target file is defined via the attributes of the export entry and SQL commands in the sub-folder **SQL Commands - DataExport** of the export entry, and the SELECT statements of the SQL queries defined in the SQL commands.

For the definition of SQL queries, the following general rules apply:

- The first property defined in the SELECT statement must be the REFSTR property of the selected object class. This first property is not written into the output file. It is only used for technical processes. If you want the REFSTR property of the object to be part of the XML output, you must add the REFSTR to the select statement two times.

- If a column name resulting from the specification in the SELECT statement starts with A_ or E_, the A_ or E_ are not written to the output file. A specification of A_PROP results in an output of PROP only. The prefix is used to define how the properties are represented in the XML.

This section lists all XML element types that can be included in the output file and how object classes and properties are represented in the output file. An example is added for a better understanding.

👁 An XML file shall be created that contains information about the as-is architecture of the company that is affected by a defined project. Both applications and business processes defined in the as-is architecture of the project shall be listed.

The ADIF export offers a mechanism to vary the definition of the parent that the information is exported for during runtime of the export execution. For more information, see Configuring Import Dependent on Parameters. To keep the following example simple, this mechanism is not used but the data is exported for an example project called "Consolidate Trading Applications".

Root Element

The XML root element of the XML is defined in the attribute section of the export entry in the attribute **XML Root Tag**. The XML root element cannot have any attributes.

👁 The root element for the example XML output is specified as "ProjectArchitecture":
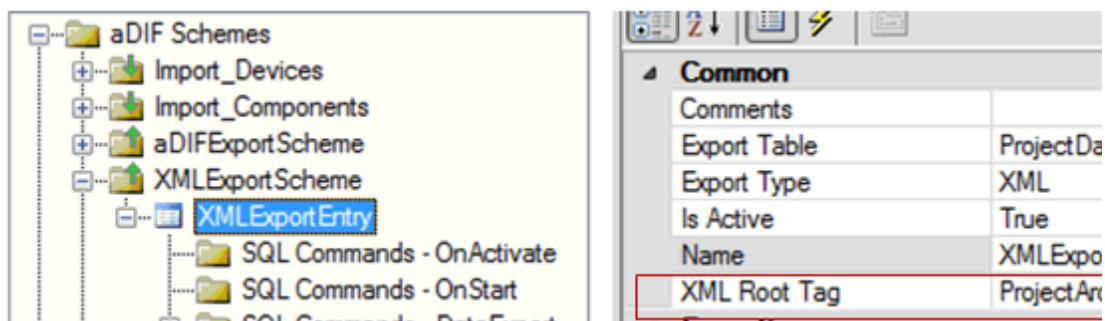
The resulting XML output is:

```
<?xml version="1.0" encoding="utf-8"?>
<ProjectArchitecture>
</ProjectArchitecture>
```

Class Elements

Class XML elements are added to the XML output on execution of the SQL commands in the folder **SQL Commands - Data Export** of the XML export entry.

For each record found by the SELECT statement defined for the SQL command, an XML element is added to the XML output. The XML element name is specified with the attribute **XML Tag Name** of the SQL command. The properties defined in the SELECT statement of the SQL command are written as XML attributes in the XML elements. For each property, an XML attribute with the name of the alias specified for the property in the SELECT statement is added to the XML element.

When specifying the alias names for the SELECT statement, you must take the following rules into account:

- The alias name must not contain whitespaces or any special characters except - or _.

- If the alias name starts with A_, the resulting attribute name will be the alias string without the A_.

- If the alias name starts with E_, the property will not be included as XML attribute in the class XML element, but as a property XML element subordinate to the class XML element.

- The first property defined in the SELECT statement is not written to the output file. For technical reasons, the first property defined in the SELECT statement must specify the REFSTR of an object.

To add information about the applications defined in the as-is architecture of the project to the XML output, an SQL command is added to the **SQL Commands - DataExport** folder of the XML export entry. The name of the XML element is set to Application with the attribute **XML Element Tag** of the SQL command.

The JOIN and WHERE conditions of the SQL query lead to the selection of applications assigned to the project 'Consolidate Trading Applications' only. The SELECT statement of the SQL query of the SQL command defines the XML attributes to be added to the XML element Application, with the REFSTR property of the object class Application added as the first property. The SQL query:

```
SELECT app.REFSTR, app.NAME "Name", app.VERSION "Version",
app.STARTDATE "StartDate", app.ENDDATE "EndDate",

(SELECT COUNT(*) FROM RELATIONS rel WHERE rel.FROMREF = app.REFSTR
AND rel.PROPERTY ='ApplicationGroups') AS "GroupAssignment"

FROM APPLICATION app

INNER JOIN PROJECT_ARCH pa ON pa.OBJECT = app.REFSTR

INNER JOIN PROJECT pro ON pro.REFSTR = pa.PROJECT

WHERE pro.NAME ='Consolidate Trading Applications'
```

Generates the following XML output:

```
<?xml version="1.0" encoding="utf-8"?>

<ProjectArchitecture>
```

```
<Application Name="Summit" Version="3.1" StartDate="2003.03.28"
EndDate="2016.01.16" GroupAssignment="2"/>

<Application Name="GenLManager" Version="1.5" StartDate="2010.05.29"
EndDate="2016.05.03" GroupAssignment="0"/>

</ProjectArchitecture>
```

The values of the XML attributes are the string values of the properties for the current record. The conversion follows the defined rules:

- The ADIF export scheme allows rules to be configured for the conversion of property values into strings. The rules are configurable in the attributes of the XML export entry (this means that the same rules apply to all XML elements in the export file).

You can define date and number formats and a string to be written to the output file for boolean `False` and boolean `True` values. For an overview of the attribute settings in the XML export entry, see Configuring Export to a Target XML File.

- Data types of the properties of Alfabet object classes are automatically mapped to the right format type and processed according to the settings in the export entry attributes.

- You can define that numeric or string values shall be converted to boolean by setting the data type as a suffix in brackets in the alias specification for the property. To be processed as boolean, the property alias must be specified as

```
<alias>[boolean]
```

and the property must return either of the following values

- The string `True` or `False`.

- An integer. If the integer is equal to 0, it is regarded as boolean `False`, all other integers are regarded as boolean `True`.

- The attribute **Write Empty Attribute Values** of the SQL Command defines whether XML attributes are written without values into the XML output if a property is not filled. Setting the attribute **Write Empty Attribute Values** to `True` results in an output of:

```
<Application Name="Summit" Version="3.1"IsVariant="" />
```

Setting the attribute to `False` will result in the same XML element to be written as:

```
<Application Name="Summit" Version="3.1"/>
```

In the example, the number of application groups that the application is assigned to is written in the XML attribute `GroupAssignment`. If the data shall only be used to find out whether an application is assigned to at least one application group, the `SELECT` statement of the Alfabet query can be changed to convert the integer that is resulting from the sub-`SELECT` statement to a boolean value:

```
SELECT app.REFSTR, app.NAME "Name", app.VERSION "Version",
app.STARTDATE "StartDate", app.ENDDATE "EndDate",
```

```
(SELECT COUNT(*) FROM RELATIONS rel WHERE rel.FROMREF = app.REFSTR
AND rel.PROPERTY ='ApplicationGroups') AS
"GroupAssignment [boolean] "

FROM APPLICATION app

INNER JOIN PROJECT_ARCH pa ON pa.OBJECT = app.REFSTR

INNER JOIN PROJECT pro ON pro.REFSTR = pa.PROJECT

WHERE pro.NAME ='Consolidate Trading Applications'
```

Generates the following XML output with `GroupAssignment` filled with the string values for boolean `True` and `False` defined in the attributes of the export entry:

```
<?xml version="1.0" encoding="utf-8"?>

<ProjectArchitecture>

    <Application Name="Summit" Version="3.1" StartDate="2003.03.28"
    EndDate="2016.01.16"GroupAssignment="yes" />

    <Application Name="GenLManager" Version="1.5"
    StartDate="2010.05.29"
    EndDate="2016.05.03"GroupAssignment="no" />

</ProjectArchitecture>
```

You can add multiple SQL commands to the **SQL Commands - DataExport** folder of an XML export entry. Class XML elements are added to the XML output in the order of the execution of the SQL commands.

SQL commands can be hierarchically structured. You can add sub-commands to the first-level commands. In the XML output, sub-commands will lead to the addition of child XML elements for an XML element. When an SQL command is executed, the child SQL command is executed for each record of the superordinate SQL command to evaluate the child XML elements to be added to the current class XML element.

The SQL queries defined for sub-commands must have a relation to the parent XML element. This relation is established by parameters that refer to the parent XML element. A parent XML element can be referenced with the parameter @<element name>. The parameter is substituted with the REFSTR specified with the first property of the SELECT statement of the parent SQL command.

For the example, the business processes assigned to the applications shall be added as sub-elements `<ApplicationProcess/>` to the `<Application/>` XML element and the business processes assigned to the project shall be added as XML elements `<BusinessProcess/>` on the top level of the hierarchy.

For the `<BusinessProcess/>` XML elements on the top level of the hierarchy, a new SQL command is added to the **SQL Commands - Data Export** folder.

For the `<ApplicationProcess/>` XML elements, a sub-command is added to the existing SQL Command for the creation of the `<Application/>` elements.
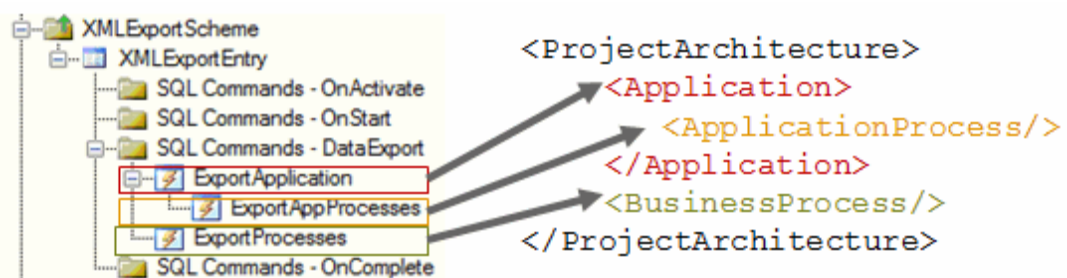
In the SQL query for the `<ApplicationProcess/>` XML elements, the applications in the upper level are referenced with the parameter `@Application`. The parameter returns the `REFSTR` of the application the parent XML element is about:

```
SELECT bp.REFSTR,bp.NAME "Name"

FROM BUSINESSPROCESS bp

INNER JOIN RELATIONS rel ON rel.FROMREF = bp.REFSTR

INNER JOIN APPLICATION app ON app.REFSTR = rel.TOREF

WHERE app.REFSTR =@ApplicationAND rel.PROPERTY ='Applications'
```

The resulting output shows the data about business processes defined by the SQL queries in the additional SQL commands:

```
<?xml version="1.0" encoding="utf-8"?>

<ProjectArchitecture>

    <Application Name="Summit" Version="3.1" StartDate="2003.03.28"
    EndDate="2016.01.16" GroupAssignment="yes">

        <ApplicationProcess Name="Equity Trading Trading" />

        <ApplicationProcess Name="Foreign Exchange Dealings" />

    </Application>

    <Application Name="GenLManager" Version="1.5"
    StartDate="2010.05.29" EndDate="2016.05.03" GroupAssignment="no">

        <ApplicationProcess Name="Custody Services" />

    </Application>

    <BusinessProcess Name="Equity Trading Trading" />

    <BusinessProcess Name="Foreign Exchange Dealings" />

    <BusinessProcess Name="Custody Services" />

</ProjectArchitecture>
```

Property Element

The properties defined in the SQL statement of an SQL command for data export for an object class can be added to the XML content as XML elements instead of being converted to XML attributes. This can be performed in two different ways:

- Defining XML element generation via a prefix in the Alias specification in the `SELECT` statement. This method generates property XML elements without XML attributes. The XML element name is identical to the Alias specification of the `SELECT` statement of the SQL command.

```
<Application Name="Summit">

    <Version>3.1</Version>

    <ObjectState>Active</ObjectState>

    <IsVariant>false</IsVariant>

</Application>
```

- Defining XML element generation via **Attribute** definitions in the **ADIF Export Entry**. This method shall be used to generate property XML elements with XML attributes or to generate property XML elements with an identical name for multiple object class properties. The name of the property XML

element generated via an **Attribute** definition is defined in the **Attribute** definition. The property XML element always has an XML attribute `Name` with the value identical to an Alias definition in the SQL statement defining the class XML element the property belongs to.

> For example the **Attribute** definition can be used to store information about object class properties in XML elements `<Property />` with an XML attribute **Name** defining the name of the object class property this XML element is created for and an XML attribute **Type** defining the data type of the object class property:

```
<Application Name="Summit">

    <Property Name="Version" Type="String">3.1</Property>

    <Property Name="ObjectState"
    Type="String">Active</Property>

    <Property Name="IsVariant" Type="String">false</Property>

</Application>
```

The following applies to both types of property XML elements:

- If you want the value of the object class property to be written to the output as CDATA, you must add a suffix `[cdata]` to the alias specification for the object class property that is defined to be written to an XML element.

- The attribute **Write Empty Element Values** of the SQL command defines whether property XML elements are written without values to the XML output if an object class property is not filled. Setting the attribute **Write Empty Element Values** to `True` results in an output of:

```
<Application Name="Summit">

    <Version>3.1</Version>

    <IsVariant />

</Application>
```

Setting the attribute to `False` will result in the same XML element to be written as:

```
<Application Name="Summit">

    <Version>3.1</Version>

</Application>
```

Defining Property XML Elements via the Alias Specification

If you want properties of an object class that are specified in the `SELECT` statement of an SQL command to be written in the file as an XML sub-element rather than XML attributes of the class XML element, you must define an alias name starting with E_ for the property in the `SELECT` statement. The value of the property is then written as data content to a child XML element of the class. The name of the child XML element is identical with the alias name for the property in the query without the prefix E_.

> In the example, the object class property `Description` of the object class `Application` shall be included in the output. As the description can be a long text and may contain special characters and line breaks, the data shall be added as a property XML element instead of an XML attribute of the `<Application/>` XML element and a `CDATA` element shall be used to wrap the

content of the description. The Description property is added to the `SELECT` statement of the SQL query as follows:

```
SELECT app.REFSTR,app.NAME "Name",app.VERSION
"Version",app.STARTDATE "StartDate[date]",app.ENDDATE
"EndDate",app.VARIANT "IsVariant",(SELECT COUNT(*) FROM RELATIONS
rel WHERE rel.FROMREF = app.REFSTR AND rel.PROPERTY
='ApplicationGroups') AS "GroupAssignment[boolean]",app.DESCRIPTION
"E_Description[cdata]"

FROM APPLICATION app
```

The resulting output for one of the applications in the output file is:

```
<Application Name="Summit" Version="3.1" StartDate="2003.03.28"
EndDate="2016.01.16" GroupAssignment="yes">

    <Description><![CDATA [Unparalleled Document Processing and Check
    Imaging Solution] ]></Description>

    <ApplicationProcess Name="Equity Trading Trading" />

    <ApplicationProcess Name="Foreign Exchange Dealings" />

</Application>
```

Defining Property Elements via Attribute definitions

If you want properties of an object class that are specified in the `SELECT` statement of an **SQL Command** to be written in the file as a XML sub-element of the class XML element with own XML attributes, you must do the following:

1) In the **SQL Command** exporting data for the object class, add all properties that you want to add as XML elements to the output to the `SELECT` statement.

2) For each property to be exported as XML element, right-click the folder **Attributes** of the export entry and select **Create Attribute**. In the attribute window of the new **Attribute** entry, define the following:

   - **XML Tag**: Enter the name of the XML Element.

   - **XML Type** : Select `Element` from the drop-down list.

   - **Export Column** : Enter the Alias name in the `SELECT` statement of the **SQL Command** for the object class property that this XML element shall store.

   - **Caption** : Enter the value for the XML attribute **Name** of the XML element.

   - **Data Type**: Enter the type of data stored in the XML element as value.

3) To add an XML attribute to a new property XML element, right-click the **Attribute** in the explorer and select **Create Xml Attribute**. In the attribute window of the new **XML Attribute** entry, define the following:

   - **Name**: Enter the name of the XML attribute.

   - **Value**: Enter the value for the XML attribute.

In the example, information about applications shall be exported, including the English and German version of the `Name` and `Description` object class properties. All object class properties shall be stored as XML child elements `<Property / >` of the `<Aplication />` element. In

addition, the information about the language culture of the exported property shall be added to the export.

The SQL Command for export of the application data includes both language versions of the `Name` and the `Description` property of the applications:

```
SELECT app.REFSTR,app.NAME +'' + app.VERSION As
"NAME_EN",app.NAME_1031 +'' + app.VERSION As "NAME_DE",
app.DESCRIPTION, app.DESCRIPTION_1031

FROM APPLICATION app
```

For each object class property an **Attribute** entry is created:
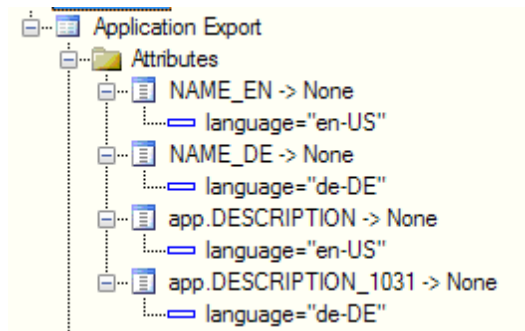
For example the **Attribute** entry for the English version of the `Name` property is defined as:
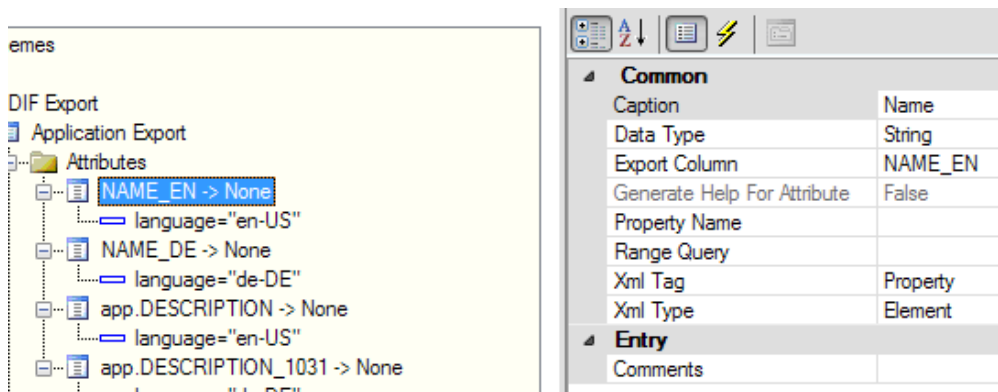
is defined as:

The **Attribute** entry for the German version of the `Name` property has the same **Caption** defined as for the English version. The difference between the two **Attribute** entries is limited to mapping to a different part of the `SELECT` statement with the **Export Column** definition and a different value definition for the XML attribute **language**.

Accordingly, the XML output of the export includes two `<Property/>` XML elements with the XML attribute **Name** set to the value `Name` and two `<Property/>` XML elements with the XML attribute **Name** set to the value `Description`. The two `<Property/>` XML elements with identical **Name** differ in the value of the XML attribute **language**. The content of the `<Property/>` XML element includes the respective translation of the applications' name and description:

```
<Application>

    <Property Name="Name" language="en-US">Customer Opinion DB
    6.1.4</Property>

    <Property Name="Name" language="de-DE">Kundenmeinungs-DB
    6.1.4</Property>
```

```
            <Property Name="Description" language="en-US">Database for
            storing results of customer surveys</Property>

            <Property Name="Description" language="de-DE">Datenbank zum
            Speichern der Ergebnisse von Kundenabfragen</Property>

        </Application>
```

Sequence Element

Sequence elements are simple XML elements without XML attributes that serve as containers for all XML elements created on execution of an SQL Command. A sequence XML element is defined in the attribute section of the SQL Command with the attribute **Sequence XML Tag Name**.

On the top level of the example, both `<Application/>` and `<BusinessProcess/>` XML elements are added. To further structure the content, the `<Application/>` XML elements are written to a sequence XML element `<ProjectApplications/>`:

The `<BusinessProcess/>` XML elements are written to a sequence XML element `<ProjectProcesses/>`. The resulting XML now looks like this:

```
<?xml version="1.0" encoding="utf-8"?>

<ProjectArchitecture>

    <ProjectApplications>

        <Application Name="Summit" Version="3.1"
        StartDate="2003.03.28" EndDate="2016.01.16"
        GroupAssignment="yes">

            <Description>Unparalleled Document Processing and Check
            Imaging Solution</Description>

            <ApplicationProcess Name="Equity Trading Trading" />

            <ApplicationProcess Name="Foreign Exchange Dealings" />

        </Application>

        <Application Name="GenLManager" Version="1.5"
        StartDate="2010.05.29" EndDate="2016.05.03"
        GroupAssignment="no">

            <ApplicationProcess Name="Custody Services" />

        </Application>

    </ProjectApplications>

    <ProjectProcesses>

        <BusinessProcess Name="Equity Trading Trading" />

        <BusinessProcess Name="Foreign Exchange Dealings" />
```

```
                    <BusinessProcess Name="Custody Services" />

          </ProjectProcesses>

        </ProjectArchitecture>
```

Configuring Export to Comma-Separated Data Files

Data can be exported to comma-separated formats to files with the extension.txt or.csv.

Data export is triggered by an SQL `SELECT` statement. The result of the SQL query is a data table that is written to the file as follows:

- For each row in the table there is a line in the file. The first line contains the table headers, and the subsequent lines contain the data records found via the SQL query.

- The separator for data columns within a data record is configurable in the ADIF export scheme.

- The data column headers in the first row are identical to the column names resulting from the `SELECT` statement of the query.

⚠️ Data that contains embedded line breaks cannot be exported correctly to CSV files. A line break is interpreted as the start of a new record. If you would like to export data containing line breaks, select a different output format (for example, export to an XML file). Line breaks can be embedded in properties of the type `Text`. Usually, descriptions provided for objects are of the data type `Text`.

👁 For example, the following SQL query:

```
SELECT app.REFSTR, app.NAME, app.VERSION, app.VARIANT

FROM APPLICATION app

WHERE app.NAME LIKE'A%';
```

Results in the following data set in a tabular format (left) and in CSV format (right):

REFSTR'	NAME	VERSION	VARIANT
76-2576-0	ALLFiance PISA	2.9	
76-2704-0	ARS BV	1.0	
76-2708-0	ARS AM	1.0.1	
76-2787-0	Administrative General Ledger	1.0	
76-2790-0	alfabet SITM	2.0	
76-2796-0	ARBI	1.2.1	
76-2797-0	ARIS	6.2.1	
76-2801-0	Aviation DB	1.0.2	
76-3217-0	ACCOUNT	1	
76-3219-0	Announcements for Federal Reserve	1.0	
76-3268-0	AF Good Buy	3.0	
76-3269-0	AF Enterprise ConTrol	4.0	
76-3275-0	AF Enterprise ConTrol	3.1	
76-3276-0	AF Good Buy	2.0	
76-3277-0	AF HR Online	2.2	
76-3278-0	AF WorkPortal	1.0	
76-3285-0	AF HR Online EU	2.2 Var.	x
76-3286-0	AF HR Online US	2.2 Var.	x
76-3289-0	AVS	4.2	
76-3306-0	AF HR Online	3.0	
76-3329-0	ACCOUNT	1.2	
76-3339-0	Asset Management	6.0	

```
REFSTR,NAME,VERSION,VARIANT
76-2576-0,ALLFiance PISA,2.9,
76-2704-0,ARS BV,1.0,
76-2708-0,ARS AM,1.0.1,
76-2787-0,Administrative General Ledger,1.0,
76-2790-0,alfabet SITM,2.0,
76-2796-0,ARBI,1.2.1,
76-2797-0,ARIS,6.2.1,
76-2801-0,Aviation DB,1.0.2,
76-3217-0,ACCOUNT,1,
76-3219-0,Announcements for Federal Reserve,1.0,
76-3268-0,AF Good Buy,3.0,
76-3269-0,AF Enterprise ConTrol,4.0,
76-3275-0,AF Enterprise ConTrol,3.1,
76-3276-0,AF Good Buy,2.0,
76-3277-0,AF HR Online,2.2,
76-3278-0,AF WorkPortal,1.0,
76-3285-0,AF HR Online EU,2.2 Var.,yes
76-3286-0,AF HR Online US,2.2 Var.,yes
```

Configuring Export to a Target CSV or TXT File

The following configuration within an ADIF export scheme is required for data export from the Alfabet database to a comma-separated format file:

- Create an export entry for the specification of the target export file.

- Define the SQL query for the definition of the content of the file in the sub-folder **SQL Commands - DataExport** of the export entry.

> ⚠ You are not allowed to define more than one SQL command of the type `Da-taExport` for export to a CSV or TXT file.

The workflow for the definition of the export entry is described below.

Optionally, you can do the following:

- Define additional SQL commands to define complex export scenarios. For more information, see Configuring Logging Parameters.

- Structure the export entries in an export set. For more information, see Automatic Creation of Multiple CSV Export Entries in an Export Set.

To create an export entry for export to CSV or TXT files:

1) In the explorer of the ADIF configuration interface, right-click the export scheme and select **Create Entry**. A new export entry node is added to the explorer as sub-node of the ADIF export scheme.

2) In the explorer, click the new export entry node.

3) In the attribute window on the right, set the following attributes for the export entry:

- **Name**: Change the default name of the entry to a meaningful name. The **Name** is used to identify the export entry in technical processes. It must be unique and should not contain whitespaces or special characters.

- **Comments**: Optionally, enter a comment that provides information about the export for internal use.

- **IsActive** : Select `True` to activate the execution of the entry during export.

- **Export Table**: Enter a name for the target file for the export. The export file will be created during export. It must have the extension.txt or.csv

- **Export Type** : Select `CSV`.

- **Delimiter**: Select the delimiter that shall be used in the export file between the data entries within a data set. Each data set is written to a separate row.

> ⓘ If the value for a data field contains the delimiter, the value will be written in quotation marks.

- **Encoding**: Select the encoding for the data in the export file from the drop-down list.

- **Boolean'False'** : Select the string value that shall be written to the export file when an attribute of the data type Boolean is set to `False`. The default is `False`.

When a boolean property is not mandatory in the Alfabet meta-model and therefore set to NULL for some objects, NULL is not interpreted as `False` but as an empty data field. If you want NULL to be treated as `False`, you must configure your SQL query accordingly.

- **Boolean'True'** : Select the string value that shall be written into the export file when an attribute of the data type Boolean is set to `True`. Default is `True`.

- **Number Decimal Digits**: Define the number of digits to be defined in the output for the decimal part of numbers. The default is 2. For example, two and a half will be written as 2.5.

- **Number Decimal Separator**: Define the decimal separator for numbers. The default is a period.

- **Number Group Separator**: Define the separator for number groups. The default is a comma. For example, three thousand will be written as 3,000.

- **Date Format**: Define the format for dates written to the export file. The default is dd/MM/yyyy.

4) In the explorer, expand the export entry node of the new export entry.

5) Right-click the folder **SQL Commands - Data Export** and select **Create SQL Command**. A new SQL command node is added to the explorer.

6) Click the new SQL command node and specify the following in the attribute window:

- **Name**: Enter a meaningful name for the SQL command.

- **Comments**: Enter a comment that gives information about the functionality implemented by the SQL command.

- **Command Type** : Select `DMLStatement` in the drop-down list

- **Result Type** : Select `Undefined`.

- **Ignore Errors** : Select `True` if you want the export to be executed even if the SQL statement specified in the attribute **Text** results in an exception. Select `False` if you do not want the export to be executed in case of an error in the SQL statement.

- **Is Active** : Select `True` to activate the SQL command. Select `False` to deactivate the SQL command.

- **Text** : Define an SQL query with a `SELECT` statement that returns a data set. The resulting data set is exported to the target file specified in the export entry. Either write the SQL query directly into the attribute field or click the **Browse** button to open an editor for the definition of the SQL query. Define the SQL query in the **SQL Text** tab of the editor.

Automatic Creation of Multiple CSV Export Entries in an Export Set

Export entries can be located directly under the ADIF export scheme element or as sub-elements of an export set. Defining export entries within an export set is useful to structure the ADIF export scheme. Additionally, the definition of an export set can be used to create multiple export entries in a batch process when defining the ADIF scheme in the ADIF configuration interface.

If you want to export data about multiple Alfabet object classes to a comma-separated format, you can select the object classes during creation of an export set via the ADIF configuration interface. One export entry per selected class is added automatically to the export set with a preconfigured SQL query for data export that selects all properties of all objects of the selected object class for export.

To create multiple export entries within an export set:

1) In the explorer of the ADIF configuration interface, right-click the ADIF export scheme to that you want to add an export set and select **Create Export Set for Classes (CSV, Excel)**. A class selector opens.

2) In the class selector, select the checkbox of the classes that you want to include in the export.

3) Click **OK** to save your changes. An export set with one export entry for each selected class is added to the explorer. For each export entry, an SQL command is predefined in the folder **SQL Commands - DataExport** with an SQL query that finds all objects of the selected class and exports all properties of the selected class.

4) Select the new export set in the explorer, and edit the following in the attribute window:

   - **Name**: Enter a meaningful name for the export set.

   - **Comments**: Enter a comment that provides information about the functionality implemented in the export set.

   - **Is Active** : Select `True` to activate the execution of all export entries within the export set. Select `False` to deactivate the execution.

5) For each export entry, select the export entry in the explorer and change at least the following attributes:

   - **Export Type** : Select `CSV` in the drop-down list.

   - **Export Table** : Change the file extension to either `csv` or `txt`. Optionally, you can change the file name.

   Optionally, you can specify all other attributes. A description of the attributes of an export entry is provided in the section Configuring Export to a Target CSV or TXT File.

6) For each export entry, navigate to the SQL command located in the **SQL Commands - DataExport** folder and modify the preconfigured command according to your requirements. For more information about the configuration of the SQL command, see Configuring Export to a Target CSV or TXT File.

   ⓘ If you want to configure complex export scenarios, you can add additional SQL commands to the export entry. For more information about the definition of SQL commands within an export entry, see Configuring Logging Parameters.

7) Optionally, reorder the sequence of export entries in the export set. Click the **Browse** button in the **Entries** attribute to open a list of all export entries in the export set. Click one of the child nodes in the list and click the **Up/Down** buttons in the upper-right corner to move the selected element in the list. Once the export entries are correctly sorted, click **OK** to apply your changes to the ADIF export scheme.

Configuring Export to Microsoft® Excel® Files

Data can be exported to Microsoft® Excel® XLS and XLSX file formats.

Data export is triggered by an SQL `SELECT` statement. The result of the SQL query is a data table that is written to the file as follows:

- Only one data set can be exported per Microsoft® Excel® file. Data is written to the first tab of the Excel file and the tab is named "Export".

- The data columns defined with the SQL `SELECT` statement define the number and content of columns in the Microsoft® Excel sheet. Optionally, ADIF can be configured to write data into the file starting from a defined position in the number of columns.

- Each row in the table equals a line in the file. The first line contains the table headers and the subsequent lines contain the data records found via the SQL query. Optionally, ADIF can be configured to add a defined text as header into the first row of the Microsoft Excel sheet and start writing the result table with the second row. The header has a green background colour and a red text:

	A	B	C	D
1	**Application Data**			
2	REFSTR	NAME	VERSION	OBJEC
3	76-2518-0	Business EA	2.2	Active
4	76-2525-0	Groupware S	2.2	Active

- The data column headers in the first row are identical to the column names resulting from the `SELECT` statement of the query.

- Excel interprets and displays number formats, date formats, and boolean values according to the standard settings of your Microsoft® Excel® application.

- The maximum number of rows in an Excel table is restricted to 64000 for some versions of Microsoft® Excel®. Additionally, you can further restrict the number of rows in the ADIF export scheme. If the number of rows in the exported data set exceeds the number of rows allowed in a table, a new file will be created containing the remaining data. Files are numbered consecutively as <FileName>.xls, <FileName>1.xls, and so on.

- Drop-down lists with pre-defined values can be created for cells in the Excel table during export. This mechanism is relevant if data shall be processed by a user in the Excel table and re-imported into the Alfabet database after processing.

For example, the following SQL query:

```
SELECT app.REFSTR, app.NAME, app.VERSION, app.VARIANT

FROM APPLICATION app

WHERE app.NAME LIKE'A%';
```

Results in the following data set in a tabular format (left) and in Microsoft® Excel® format (right):

REFSTR	NAME	VERSION	VARIANT
76-2576-0	ALLFiance PISA	2.9	
76-2704-0	ARS BV	1.0	
76-2708-0	ARS AM	1.0.1	
76-2787-0	Administrative General Ledger	1.0	
76-2790-0	alfabet SITM	2.0	
76-2796-0	ARBI	1.2.1	
76-2797-0	ARIS	6.2.1	
76-2801-0	Aviation DB	1.0.2	
76-3217-0	ACCOUNT	1	
76-3219-0	Announcements for Federal Reserve	1.0	
76-3268-0	AF Good Buy	3.0	
76-3269-0	AF Enterprise ConTrol	4.0	
76-3275-0	AF Enterprise ConTrol	3.1	
76-3276-0	AF Good Buy	2.0	
76-3277-0	AF HR Online	2.2	
76-3278-0	AF WorkPortal	1.0	
76-3285-0	AF HR Online EU	2.2 Var.	x
76-3286-0	AF HR Online US	2.2 Var.	x
76-3289-0	AVS	4.2	
76-3306-0	AF HR Online	3.0	
76-3329-0	ACCOUNT	1.2	
76-3339-0	Asset Management	6.0	

	A	B	C	D
1	REFSTR	NAME	VERSION	VARIANT
2	76-2576-0	ALLFiance PI	2.9	
3	76-2704-0	ARS BV	1.0	
4	76-2708-0	ARS AM	1.0.1	
5	76-2787-0	Administrativ	1.0	
6	76-2790-0	alfabet SITM	2.0	
7	76-2796-0	ARBI	1.2.1	
8	76-2797-0	ARIS	6.2.1	
9	76-2801-0	Aviation DB	1.0.2	
10	76-3217-0	ACCOUNT	1	
11	76-3219-0	Announceme	1.0	
12	76-3268-0	AF Good Buy	3.0	
13	76-3269-0	AF Enterprise	4.0	
14	76-3275-0	AF Enterprise	3.1	
15	76-3276-0	AF Good Buy	2.0	
16	76-3277-0	AF HR Online	2.2	
17	76-3278-0	AF WorkPort	1.0	
18	76-3285-0	AF HR Online	2.2 Var.	WAHR
19	76-3286-0	AF HR Online	2.2 Var.	WAHR
20	76-3289-0	AVS	4.2	
21	76-3306-0	AF HR Online	3.0	
22	76-3329-0	ACCOUNT	1.2	
23	76-3339-0	Asset Manag	6.0	

Configuring Export to a Target Microsoft® Excel® File

The following configuration within an ADIF export scheme is required for data export from the Alfabet database to a Microsoft® Excel® file:

- Create an export entry for the specification of the target export file.

- Define the SQL query for the definition of the content of the file in the sub-folder **SQL Commands - DataExport** of the export entry.

> ⚠ You are not allowed to define more than one SQL command of the type `Da-taExport` for export to a target Microsoft® Excel® file.

The workflow for the definition of the export entry is described below.

Optionally, you can do the following:

- Define additional SQL commands to define complex export scenarios. For more information, see Configuring Logging Parameters.

- Structure the export entries in an export set. For more information, see Automatic Creation of Multiple CSV Export Entries in an Export Set.

To create an export entry for export to Microsoft Excel files:

1) In the explorer of the ADIF configuration interface, right-click the export scheme and select **Create Entry**. A new export entry node is added to the explorer as a sub-node of the ADIF export scheme.

2) In the explorer, click the new export entry node.

3) In the attribute window on the right, set the following for the export entry:

- **Name**: Change the default name of the entry to a meaningful name. The **Name** is used to identify the export entry in technical processes. It must be unique and should not contain whitespaces or special characters.

- **Comments**: Optionally, enter a comment that provides information about the export for internal use.

- **IsActive** : Select `True` to activate the execution of the entry during export.

- **Export Table**: Enter a name for the target file for the export. The export file will be created during export. It must have the extension.xls or.xlsx. The file extension must be identical to the extension selected with the attribute **Export Type**.

- **Export Type** : Select `XLS` to export to a Microsoft® Excel® 2003 format or `XLSX` to export to a Microsoft® Excel® 2010 format.

- **Excel Max Records**: Define the maximum number of records that may be added to the target file. If the number of records in the data set exceeds the maximum number of records, the remaining data will be written to a new file named <FileName><running number>.<extension>. You can define any number of records higher than 100 or -1 to write data to the target file without limitations in the number of records.

  (i) Make sure that the configured number corresponds to the technical limits of the Microsoft® Excel® version used. Some Excel® applications cannot read tables with more than 64000 records.

- **Header Text**: If you want a general header to be displayed above the data set in the Microsoft Excel sheet, enter the header text. You can click the ⊡ button in the attribute field to open a text editor. If you include a line break, the line break is also available in the header text in the Microsoft Excel file.

- **Export Start Column**: Enter the number of the first column of the exported data set in the Microsoft Excel sheet. The first column in the sheet has the number 0.

  ◎ For example to generate a Microsoft Excel sheet with three empty columns before the exported data, set the Export Start Column attribute to 3:

	A	B	C	D	E	F
1				REFSTR	NAME	VERSION
2				76-2518-0	Business EA	2.2
3				76-2525-0	Groupware S	2.2
4				76-2538-0	Mafo-Portal	2.6

4) In the explorer, expand the export entry node of the new export entry.

5) Right-click the folder **SQL Commands - Data Export** and select **Create SQL Command**. A new SQL command node is added to the explorer.

6) Click the new SQL command node and specify the following attributes in the attribute window:

- **Name**: Enter a meaningful name for the SQL command.

- **Comments**: Enter a comment that provides information about the functionality implemented by the SQL command.

- **Command Type** : Select `DMLStatement` in the drop-down list

- **Result Type** : Select `Undefined`.

- **Ignore Errors** : Select `True` if you want the export to be executed even if the SQL statement specified in the attribute **Text** results in an exception. Select `False` if you do not want the export to be executed in case of an error in the SQL statement.

- **Is Active** : Select `True` to activate the SQL command. Select `False` to deactivate the SQL command.

- **Text** : Define an SQL query with a `SELECT` statement that returns a data set. The resulting data set is exported to the target file specified in the export entry. Either write the SQL query directly in the attribute field or click the **Browse** ![button] button to open an editor for the definition of the SQL query. Define the SQL query in the **SQL Text** tab of the editor.

Creating Drop-Down lists in the Exported Excel File

Fields in the resulting export file can be written as a drop-down list field during export with a set of defined values. If the data is exported to be updated within the file and re-imported via an ADIF import scheme into the Alfabet database, this mechanism ensures that the person editing the data does not enter invalid values.

For example:

- A property is based on an enumeration. The values defined for the enumeration can be selected from the drop-down list. No invalid values can be produced by typos or by a user adding values not included in the enumeration.

- A property stores a link to another object in the Alfabet database. The name of the objects that may be target of the link are defined in a drop-down list. For example for assigning components to a component category, the component catagories defined in the Alfabet database are displayed in the list. The person editing the data can specify relations without opening the Alfabet user interface to find out which component categories are available.

To define a drop-down list in the output Microsoft Excel file of the export:

1) Right-click the folder **Attributes** of the export entry and select **Create Attribute**. In the attribute window of the new **Attribute** entry, define the following:

- **Export Column** : Enter the Alias defined in the `SELECT` statement of the **SQL Command** for data export for the column in the exported dataset the drop-down list shall be applied to.

- **Range Query**: Enter a native SQL query that returns the data to be displayed in the drop-down list. For each data set found by the query a row will be added to the drop-down list. All data defined in the SELECT statement of the **Range Query** is displayed in the row of the drop-down list. If the `SELECT` statement has multiple arguments, the values are written into the row with a whitespace as separator.

  > (i) If the drop-down list shall display values derived for example from an enumeration, the **Range Query** can be defined to return text strings:

  ```
  SELECT 'Active' UNION ALL SELECT 'Retired' UNION ALL SELECT
  'Plan'
  ```

  If the drop-down list shall return a list of available objects in the Alfabet database, the **Range Query** can read the data directly from the database. Please note that the drop-down list must enable the person processing the exported data to identify the object unambiguously. For example, a drop-down list that allows to select applications must include application name and version, because uniqueness is only implemented for the combination of both:

  ```
  SELECT NAME, VERSION

  FROM APPLICATION

  ORDER BY NAME
  ```

- **Comments**: Optionally, enter a comment that provides information about the definition for internal use.

Automatic Creation of Multiple Excel Export Entries in an Export Set

Export entries may be located directly below the ADIF export scheme element or as sub-elements of an export set. Defining export entries within an export set is useful to structure the ADIF export scheme. Additionally, the definition of an export set can be used to create multiple export entries in a batch process when defining the ADIF scheme in the ADIF configuration interface.

If you want to export data about multiple Alfabet object classes to a Microsoft® Excel® file, you can select these object classes during creation of an export set via the ADIF configuration interface. One export entry per selected class is added automatically to the export set with a preconfigured SQL query for data export that selects all properties of all objects of the selected object class for export.

To create multiple export entries within an export set:

1) In the explorer of the ADIF configuration interface, right-click the ADIF export scheme for which you want to add an export set and select **Create Export Set for Classes (CSV, Excel)**. A class selector opens.

2) In the class selector, select the checkbox of the object classes that you want to include in the export.

3) Click **OK** to save your changes. An export set with one export entry for each selected object class is added to the explorer. For each export entry, an SQL command is predefined in the folder **SQL**

Commands - DataExport with an SQL query that finds all objects of the selected object class and exports all object class properties of the selected object class.

4) Select the new export set in the explorer, and edit the following in the attribute window:

- **Name**: Enter a meaningful name for the export set.

- **Comments**: Enter a comment that provides information about the functionality implemented in the export set.

- **Is Active** : Select `True` to activate the execution of all export entries within the export set. Select `False` to deactivate the execution.

5) Optionally, you can modify the attributes of the export entries according to your requirements. A description of the attributes of an export entry is provided in the section Configuring Export to a Target CSV or TXT File.

6) For each export entry, navigate to the SQL command located in the **SQL Commands - DataExport** folder and alter the preconfigured command according to your requirements. For more information about the configuration of the SQL command, see Configuring Export to a Target CSV or TXT File.

> ⓘ If you want to configure complex export scenarios, you can add additional SQL commands to the export entry. For more information about the definition of SQL commands in an export entry, see Configuring Logging Parameters.

7) Optionally, reorder the sequence of export entries in the export set. Click the **Browse** button in the **Entries** attribute to open a list of all export entries in the export set. Click one of the child

   nodes in the list and click the **Up/Down** buttons in the upper-right corner to move the selected element in the list. Once the export entries are correctly sorted, click **OK** to apply your changes to the ADIF export scheme.

Configuring SQL Commands for Optional Export Enhancements

The basic export configuration described in the section Basic Configuration of Data Export can be enhanced by defining additional SQL commands. When creating ADIF export schemes via the ADIF configuration explorer, folders for SQL commands are automatically added to the explorer when SQL commands can be defined for the element. The folders also define the type of SQL command.

> ⓘ In the XML of the ADIF export scheme each SQL command element has an attribute **Type**. A separate folder named **SQL Commands - <Type>** is added to the explorer for each permissible type. When SQL Commands are defined in a folder, the **Type** attribute is automatically set in the XML definition. When specifying SQL Commands in an XML editor, the **Type** attribute must be defined manually.

The following SQL Command types are available:

Type	Location in ADIF Scheme	Description
`OnActivate`	Export Sets and Export Entries	SQL commands of the type `OnActivate` define conditions for the execution of the element they are defined for. For more information, see Configuring the Conditional Execution of Parts of the Import Scheme. **Note** : Changes triggered by `OnActivate` commands are not rolled back if the option **Commit After Run** is set to `False` for the import scheme.
`DataExport`	Export Entries	SQL commands of the type `DataExport` define the data that shall be written to the target export file or export database table. Each export entry must include one SQL command of the type `DataExport`. The permissible number of SQL commands for data export depends on the target data format. For more information, see Basic Configuration of Data Export.
`OnStart`	Export Entries	SQL commands of the type `OnStart` are executed prior to the SQL command of the type `DataExport` and SQL commands of the type `OnComplete` are executed after the SQL command of the type `DataExport`. The commands allow to execute special operations on the Alfabet database prior to or after export. The SQL commands can be defined for the following purposes: • Definition of customized debug information written to the log file during execution of exports. This configuration is described in the section Configuring Log Message Content. • Manipulation of the Alfabet database prior and/or after data export. The definition of SQL command for this purpose is described in this section.
`OnComplete`	Export Entries	

One of the main features of the ADIF interface is the flexibility in configuration. The ADIF export scheme can not only be configured to simply read existing data from a Alfabet database table to an export file or an external database table, but you can configure ADIF to perform additional operations on the Alfabet database or, in case of export to an external database, on the external database table prior to or after the actual data export. SQL queries of the **Command Type** `DDLStatement` or `DMLStatement` can be defined in any number and order required.

> If SQL queries are not sufficient to implement the desired export, custom DLLs can be ordered from Software AG to provide functionality. The custom code is added to the ADIF export scheme via an SQL command with the **Command Type** `StoredProcedure`. The correct specification of configuration elements and attributes depends on the custom code. A description is delivered with the code on an individual basis.
>
> Based on the order of execution of SQL statements described below, stored procedures are executed in the same way as DML statements.

When configuring SQL commands for an ADIF export scheme, the order of execution must be taken into account. The elements of an ADIF scheme are executed in the following order:

- The SQL commands `OnActivate` are evaluated first and the respective elements of the ADIF schemes are included or excluded from the execution schedule.

- All SQL commands of the type `OnStart` and the **Command Type** `DDLStatement` that target the Alfabet database are executed in the order they are specified in the ADIF export scheme.

- All export entries and export data sets specified in the ADIF scheme are then processed in the order specified in the ADIF export scheme.

  - Export data sets for export to external databases are processed as follows:

    - The connection to the external database is established.

    - All DDL statements of SQL commands of the type `OnStart` targeting the external database configured for the export entries in the database export set are executed in the order they are defined in the export set.

    - The export entries are processed in the order they are configured in the database export set as follows:

      - All DML statements of SQL commands of the type `OnStart` are executed in the order they are specified in the export entry.

      - The SQL command of the type `DataExport` is executed.

      - All DML statements of SQL commands of the type `OnComplete` are executed in the order they are specified in the export entry. These SQL commands can target either the Alfabet or the target database.

    - All DDL statements of SQL commands of the type `OnComplete` targeting the external database configured for the export entries in the database export set are executed in the order they are defined in the export set.

  - All other export entries are processed as follows independent of the location inside or outside an export set:

    - All DML statements of SQL commands of the type `OnStart` are executed in the order they are specified in the export entry.

    - The SQL commands of the type `DataExport` are executed.

    - The SQL commands of the type `OnComplete` are executed in the order they are specified in the export entry.

- All DDL statements of SQL commands of the type `OnComplete` targeting the Alfabet database configured for the export entries in the ADIF export scheme are executed in the order they are defined in the ADIF export scheme.

This section describes in general how to define an SQL command within an **SQL Commands - OnStart** or **SQL Commands - OnComplete** folder of the ADIF configuration explorer, and describes two use cases that may be covered by the definition of SQL commands:

- *Use Case: Update Existing Data Records in a Target Database*

- *Use Case: Defining Export-Related Custom Properties*

- *Use Case: Data Restructuring Prior to Export Using Temporary Tables*

Creating SQL Commands

To specify an SQL command to be executed on the Alfabet database or an external target database prior to or after data export:

1) In the explorer of the ADIF configuration interface, right-click the folder **SQL Commands - OnStart** or **SQL Commands - OnComplete** that is automatically added to the explorer upon the creation of an **Export Entry** node.

2) In the context menu, select **Create SQL Command**. A new **SQL Command** element is added as child node of the folder.

3) Click the new SQL command node and specify the following in the attribute window:

   - **Name**: Enter a meaningful name for the SQL command.

   - **Comments**: Enter a comment that gives information about the functionality implemented by the SQL command.

   - **Command Type**: Select the type of SQL query that you want to define from the drop-down list. Select:

     - `DMLStatement` for SQL data manipulation statements like, for example, `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `MERGE`.

     - `DDLStatement` for SQL data definition statements like, for example, `CREATE`, `ALTER`, `DROP`, `TRUNCATE`, `RENAME`.

       ⚠️ DDL statements are only allowed to create, alter and drop new database tables. It is not permissible to apply those commands to standard Alfabet database tables.

   - **Result Type** : Select `Undefined`.

   - **ApplyTo**: If the SQL command is defined for export to an external database table, define the database that the SQL command is executed on:

     - `local` for execution on the Alfabet database

     - `external` for execution on the external database

   - **Ignore Errors** : Select `True` if you want the export to be executed even if the SQL statement specified with the attribute **Text** results in an exception. Select `False` if you do not want the export to be executed in case of an error in the SQL statement.

   - **Is Active** : Select `True` to activate the SQL command. Select `False` to deactivate the SQL command.

   - **Text**: Define the SQL query for the command. Either write the SQL query directly in the attribute field or click the **Browse** ⊞ button to open an editor for the definition of the SQL query. Define the SQL query in the **SQL Text** tab of the editor.

Use Case: Update Existing Data Records in a Target Database

If you want to export data to an external database table that already contains records, you must implement mechanisms in the ADIF export scheme that ensure data integrity in the target database table. Primary key

constraints and unique key configurations of the target database completely depend on the individual configuration of the target database and can therefore not be taken into account by the ADIF data export mechanism.

During execution of the SQL command for data export, data is simply written to the external database table in new records. If this leads to key violations, export is stopped and an error message is written to the log file of the ADIF console application. Even if the data can be added to the table with new keys for each object, data that is meant to update existing object data leads to the creation of a new record instead of updating the existing one.

The ADIF export entry for export to external databases allows you to define any number of DDL and DML statements to be executed on the target database either prior to or after data export is performed.

To implement data update in existing database tables, you can, for example, create a new database table in the target database prior to data export and export the data to the new table. After data export, SQL commands can be executed that update the actual target tables in the external database, taking into account the required key constraints and other update requirements.

The new database table can then be deleted after the target tables were changed. Two mechanisms are available to delete the temporary table:

- To delete the table as a last step of the execution of the export entry and prior to the execution of the next export entry, define a DDL statement that drops the table in the **SQL Commands - OnComplete** folder of the export entry.

- To delete the table after all SQL commands in all export entries of the data set were executed, set the attribute **Delete Table After Export** to `True` for the export entry. This configuration allows you to use data from the table in SQL commands configured in other export entries executed after the current export entry during data export to the external database.

> (i) For regular exports, it might be more convenient to create a table for data export in the target database either during the first import or by mechanisms outside of ADIF and keep the table persistently. You can then set the attribute **Empty Table Before Export** in the export entry to delete the data from the database table prior to each new import of data in the external database.

If you export data primarily to a temporary table, you can only use one export entry to update multiple database tables of the external database. The number of tables changed by SQL commands in the folder **SQL Commands - OnComplete** is unlimited.
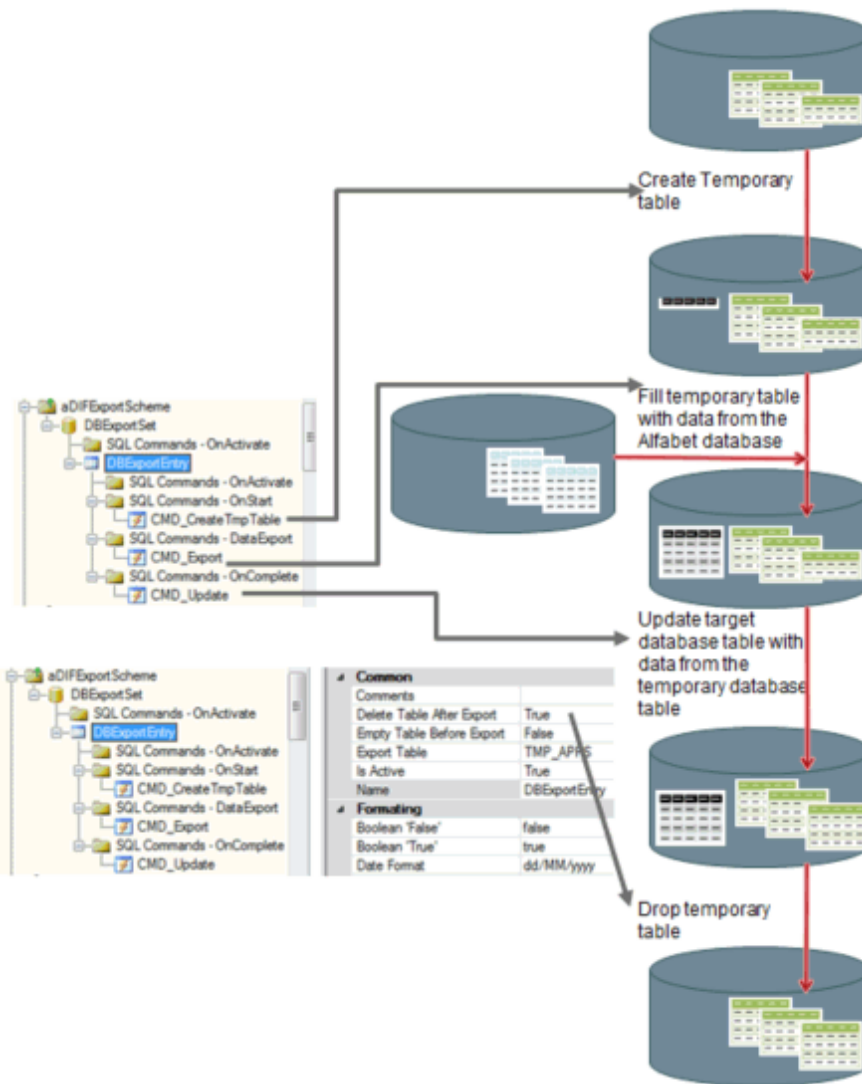
FIGURE: Overview of the configuration steps performed to update data in an external database

Use Case: Defining Export-Related Custom Properties

It may be useful to store the information about the performance of the export directly in the database table of the exported objects for the following reasons, for example:

- Configure export to include only objects that are not already marked as exported in a prior data export.

- Inform the user via the Alfabet interface whether and/or when data has been last exported.

- Use the attribute as a condition for the execution of actions via the Alfabet interface (for example, in workflow preconditions or in reports to generate a report that only shows data that was already exported).

The following configuration steps are required to store information about the export performance directly in the database table of the exported objects:

- A custom property must be added to the object class for which data is exported. The customization of object classes via customer-defined properties is part of the configuration capabilities offered by Alfabet Expand independent from the ADIF user interface. For information about how to create a custom property, see *Configuring Custom Properties for Protected or Public Object Classes* in the reference manual *Configuring Alfabet with Alfabet Expand*.

- In the export entry triggering the export, you can then add an SQL command with an `UPDATE` statement in the folder **SQL Commands - OnStart** or **SQL Commands - OnComplete** that finds the exported objects and writes the information about the export to the custom attribute. The custom attribute may contain the date that the export was performed or a boolean value that distinguishes between exported and not exported data.

⚠ Always use the configuration functionalities of Alfabet Expand to add new property columns to a database table of a Alfabet object class. It is not permissible to add a property column via a DDL statement in the SQL commands of the ADIF scheme. Existing standard Alfabet database tables must not be altered by SQL commands via ADIF.

Use Case: Data Restructuring Prior to Export Using Temporary Tables

In most exports (except the export to XML), the content of the export must be defined with a single SQL query with a `SELECT` statement. SQL provides a high flexibility in data collection in a single `SELECT` statement. Furthermore, ADIF offers additional functionality for the use cases where data must be manipulated or restructured prior to export in a way that can not be covered by defining SQL queries.

New database tables can be added to the Alfabet database prior to export to restructure the existing data or add new data derived from query results. The SQL queries in the **SQL Commands - DataExport** folder can then be defined to export data from the new database tables exclusively or in parallel with data from the usual Alfabet database tables.

⚠ New database tables should only be added temporarily to the Alfabet database. If you add a table via an SQL command to be executed on the start of the export entry, make sure that you add an SQL command to be executed on completion of the export that drops the new database table.

⚠ If the output of the export is an XML file, the data from the temporary table can only be included in the output if the first column of the export definition `SELECT` statement has the alias name "REFSTR". This first property in the `SELECT` statement will be ignored.
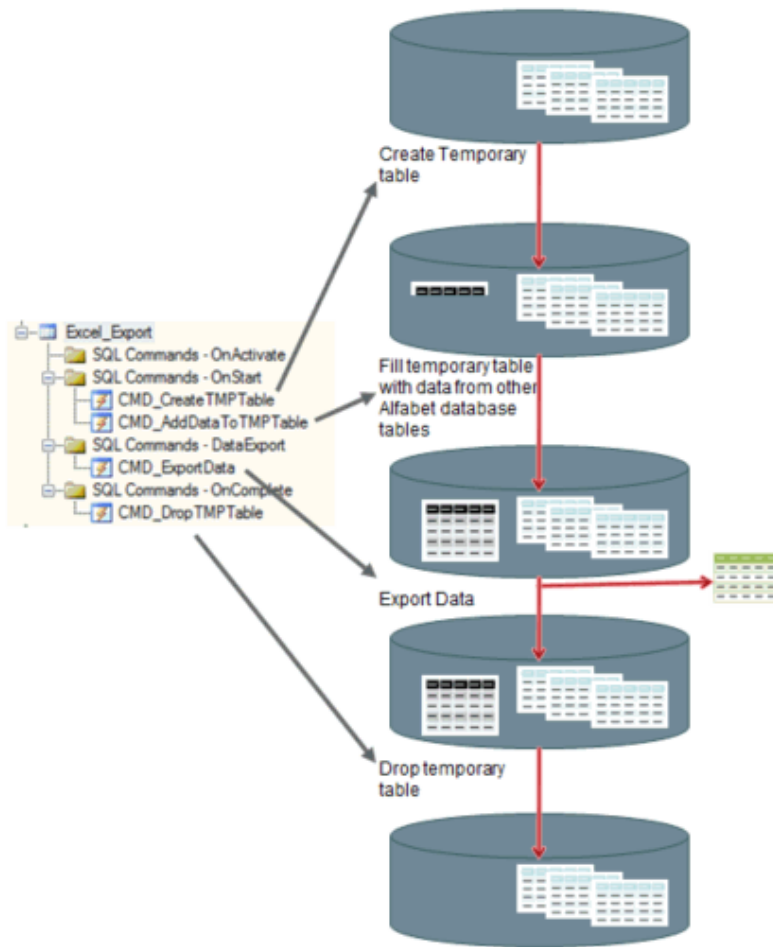
FIGURE: Creation of temporary database tables during export via SQL commands for an export entry

Configuring Execution Dependent on Current Parameters

ADIF export schemes can be configured to adapt to current states of objects in the Alfabet database. You can define conditions for the overall execution of parts of the export scheme or define conditions for the choice of object data for execution that can be set in the command line of the ADIF console application during export and can therefore vary per export.

Configuring Conditional Execution for Parts of the Export Scheme

The ADIF export scheme allows you to couple the execution of parts of the export definition to conditions. You can define SQL commands that check the Alfabet database for the availability of data and configure export sets or export entries to be executed only if a configured SQL query executed on the Alfabet database delivers either a positive or negative result.

👁 For example: Data shall be exported about all applications for which a specific indicator shows critical values. The export can be configured to be executed only if a critical value of the indicator is reached for at least one application.

Conditions can be defined for the following elements of the ADIF export scheme:

- DB Export Set

- Excel Export Set

- XML Export Set

- Export Entry

If a condition is specified for an element, this element and all child elements of the current element are only executed if the condition is met.

To specify a condition for the execution of a part of an ADIF export scheme:

1) In the explorer of the ADIF configuration interface, right-click the folder **SQL Commands - OnActivate** that is automatically added to the explorer on creation of an element.

2) In the context menu, select **Create SQL Command**. A new SQL command element is added as a child node of the **SQL Commands - OnActivate** folder.

3) Click the new SQL command node and specify the following in the attribute window:

    - **Name**: Enter a meaningful name for the SQL command.

    - **Comments**: Enter a comment that provides information about the functionality implemented by the SQL command.

    - **Apply to**: Select `local`. SQL commands of the type **OnActivate** can only be executed on the Alfabet database.

    - **Result Type**: Select one of the following in the drop-down list:

        - `PositiveCheck`: If the data set that results from the execution of the query defined with the attribute **Text** is empty, the current element of the ADIF scheme will not be executed and the message defined with the attribute **Message** will be written to the log file.

        - `NegativeCheck`: If the data set that results from the execution of the query defined with the attribute **Text** is not empty, the current element of the ADIF scheme will not be executed and the message defined with the attribute **Message** will be written to the log file.

    - **Text** : Define an SQL query with a `SELECT` statement that returns a data set. The resulting data set is checked according to the settings of the attribute **Result Type** to specify whether the ADIF element is executed. Either write the SQL query directly in the attribute field or click the **Browse** button to open an editor for the definition of the SQL query. Define the SQL query in the **SQL Text** tab of the editor.

    - **Message**: Enter a text that shall be written to the log file in case the ADIF element is not executed.

    - **Ignore Errors** : Select `True` if you want the export to be executed even if the SQL statement specified with the attribute **Text** results in an exception. Select `False` if you do not want the import to be executed in case of an error in the SQL statement.

    - **Is Active** : Select `True` to activate the check. Select `False` to deactivate the check.

The condition for the check can be defined dependent on parameters set during runtime of the ADIF import. For more information, see [Configuring Import Dependent on Parameters](#).

Configuring Export Dependent on Parameters

Export can be configured to be based on parameters defined in the command line of the ADIF console application or in the JSON body of the RESTful service call when starting an export. This allows you to reuse an ADIF export scheme for various similar exports that only vary in small aspects.

ⓘ   It is not possible to define parameters for ADIF exports executed via the user interface. Nevertheless, these exports can be executed with the default values for the parameters if these default values are defined in the ADIF export scheme. If an ADIF scheme configuration includes mandatory parameters, the execution option in the **ADIF Jobs Administration** (`ADMIN_AdifJobs`) functionality.

◎   For example, an export that is configured to update an external database with information changed in the Alfabet database can be performed with the latest change data of objects as a parameter. You can then configure the ADIF interface to export data only if the object has been changed after a specific date that is defined by a parameter. For each export, the parameter is set in the command line to the date when the last export was performed.

The following information is available:

- [Configuring an ADIF Export Scheme to Use Parameters in Export Definitions](#)

- [Defining Parameter Values On Execution of ADIF Schemes](#)

- [Defining Parameter Values During Testing of the ADIF Scheme](#)

Configuring an ADIF Export Scheme to Use Parameters in Export Definitions

Parameters have to be defined in the ADIF scheme. Only parameters that are defined in the ADIF scheme can be assigned a value during execution of the ADIF scheme.

To define the parameters for an ADIF scheme:

1)   In the explorer, click on the ADIF export scheme that you would like to start with parameters.

2)   In the attribute window, define the parameter relevant attributes:

- **Parameters Backward Compatibility Mode** : Make sure that the parameter is set to `False`. This is the default value for new ADIF schemes.

    ⓘ   ADIF schemes defined prior to Alfabet 10.4 will have the compatibility mode set to `True`. If you reset the compatibility mode for these schemes, parameter definitions must be added to the ADIF export entries in the scheme and the way parameters are defined in the SQL commands must be revised.

    Please note that ADIF schemes with a **Parameters Backward Compatibility Mode** set to `True` which contain parameter definitions cannot be scheduled for execution via the **Job Schedule** functionality.

- **Arguments Table Name** : At the start of an ADIF export, the parameter values are written to a temporary database table. The name of the temporary database table is configurable in the ADIF export scheme. By default, it is named `ADIF_ARGS`.

ⓘ For each parameter, a row is displayed in the database table with two columns:

- `ARG_NAME` for the storage of the parameter name.

- `ARG_VALUE` for the storage of the parameter value.

3) For each parameter that you would like to use in one of the export entries in the scheme, right-click the **Parameters** child node of the ADIF export scheme and select **Create Parameter** from the dropdown list.

4) Click the newly added **@Parameter** node and define the following attributes in the attribute window:

- **Name**: Define a unique name for the parameter. The parameter name must start with @ and must not include other special characters or whitespaces. It can be used in the SQL commands and attributes of the ADIF export scheme and is substituted with the current value of the parameter during execution of the ADIF export scheme.

- **Parameter Type**: Select the data type of the parameter value from the dropdown list.

    ⓘ The parameter types StringArray and ReferenceArray can be used if a query shall define that a string or the REFSTR of a referenced object shall be within an allowed range of string or REFSTR values.

- **Default Value**: Optionally, define a default value that shall be used if no parameter value is provided during execution of the ADIF scheme. Click the **Browse** 🔳 button on the right of the attribute's field to open the editor for definition of the default value. Note the following about the specification of values:

    - For the definition of dates enter the date into the **Value** field in the format defined in the culture setting of the language you are currently using to render the Alfabet Expand user interface or click the **Select** button and select a date from the calendar.

    - If the parameter is placeholder for a string and used in a query in a `WHERE` condition that checks whether a value is one of a range, the **Parameter Type** must be `StringArray`. The **Value** field in the editor of the **Default Value** attribute displays a text field for `StringArray` parameters. Define the range of allowed values in the text field with each value in a separate row. In the query refering to the array, the parameter must be added in brackets:

      ```
      WHERE app.OBJECTSTATE IN (@appState)
      ```

    - If a string value contains a single quote (like for example in O'Hara), the single quote will be automatically escaped with a second single quote when the parameter value is processed. Escaping the single quote with a second single quote is a requirement for including it into SQL queries. For security reasons, the second single quote is also added in strings to prevent SQL injection.

    - % can be used as wildcard in strings and texts. It is not allowed to define a wildcard in a value of a string array.

    - Boolean values can be selected from a dropdown list. Please note that object class properties of the type `Boolean` that are neither mandatory nor having a default value defined might be set to either true, false or NULL. If the default value is `False`, only object class property values defined as false will be returned, while NULL values will

not be returned. This behaviour can be handled via the SQL commands in the ADIF scheme.

- For ReferenceArray properties no default values can be defined.

- **Mandatory** : Select `True` if the ADIF execution should fail with an error message if no value is defined for the current execution. Select False, if the ADIF execution should be executed even if the parameter is not defined for the current execution. The parameter will then be substituted with the value defined in the attribute **Default Value**. If no default value is provided, the parameter will be substituted with `NULL`.

> ⚠️ In ADIF execution, WHERE clauses in SQL commands containing a parameter are not removed from the query if no value is provided. To make sure that the query result is meaningful if the parameter value is not provided during ADIF execution and the parameter is therefore set to NULL, you should define the query to accept NULL values. For example, to ensure that results are returned if a date compared to an application's start date is not returned, the WHERE clause should be written as:
>
> ```
> WHERE APPLICATION.STARTDATE >= @StartDate OR @StartDate IS
> NULL
> ```

5) Use the parameters defined in the **Parameters** folder in the ADIF export entries of the ADIF export scheme in the following settings according to demand:

- In the SQL queries defined for any SQL command.

- In the **Message** attribute of SQL commands of the **Result Type** `DebugMessage`.

- In the **Connection String** attribute of a database export set.

- In the **Header Text** attribute of export entries for export to Microsoft® Excel® files.

To include a parameter, substitute the actual value with the parameter name within the query or the text string. Please note that parameters are to be included without any single quotes around the parameter name. If the parameter type is requiring definition of single quotes around the value, this is handled by the software during substitution of the parameter name with the parameter value.

> 👁 For example, a query using the string parameter `@orga` may be written as:
>
> ```
> SELECT app.NAME, app.VERSION
>
> FROM APPLICATION app, ORGAUNIT org
>
> WHERE app.RESPONSIBLEORGANIZATION = org.REFSTR
>
> AND org.NAME LIKE @orga;
> ```
>
> If it is during runtime with the value `Trade%`, the query will read:
>
> ```
> SELECT app.NAME, app.VERSION
>
> FROM APPLICATION app, ORGAUNIT org
>
> WHERE app.RESPONSIBLEORGANIZATION = org.REFSTR
>
> AND org.NAME LIKE'Trade%';
> ```
>
> A debug message defined in the attribute **Message** of an SQL command of the **Result Type** `DebugMessage` to be written in the export log file during export of the application data can refer to the current organization as well:
>
> ```
> Export of applications with responsible organization @orga.
> ```

During runtime, it will be converted to:

```
Export of applications with responsible organization Trade%.
```

StringArray or ReferenceArray parameters need to be included into the query as follows:

```
WHERE APPLICATION.STATUS IN (@StringArray)
```

Defining Parameter Values On Execution of ADIF Schemes

Parameters values are defined during runtime on execution of the ADIF import or export. The parameter name is case sensitive. When defining parameters for execution of the ADIF import or export, you must refer to the parameter name exactly as written in the ADIF scheme, including the starting @. Note the following about the definition of values for the parameters:

- If a string value contains a single quote (like for example in O'Hara), the single quote will be automatically escaped with a second single quote when the parameter value is processed. Escaping the single quote with a second single quote is a requirement for including it into SQL queries. For security reasons, the second single quote is also added in strings to prevent SQL injection.

- Parameter values for data types like string or date are defined without single quotes at the beginning and end. If single quotes are required in the query for the data type, they will be automatically added by the ADIF mechanisms.

- Boolean values must be defined as 1 (true) or 0 (false):

  ```
  @MyBooleanValue 1
  ```

- String array and ReferenceArray values must be separated with \r\n:

  ```
  @MyArrayValue First\r\nSecond
  ```

Whether and how command line parameters can be defined at runtime depends on the way the ADIF scheme is executed:

- **Starting ADIF via the console application**

To specify parameters in the command line of the ADIF console application, start the console application with the command line option -<parameter name> <parameter value>. A command line option must be specified for each parameter.

> To specify the parameters @lastupdate and @orga, the command line could for example be defined as:
>
> ```
> ADIF_Console.exe -export -msalias Alfabet -alfaLoginName
> UserName -alfaLoginPassword pw1234 -scheme AppExport -
> exportfile C:\Export\Applications.zip -@lastupdate 09/06/2018
> -@orga ITGroup
> ```

> For general information about starting ADIF via the console application, see Executing ADIF via a Command Line Tool.

- **Starting ADIF via a RESTful service call to the RESTful API of the Alfabet Web Application**

Parameters are defined in the JSON body of the request in the field UserArgs returning a JSON object with one field for each parameter in the format "UserArgs": {" arg1name ":" arg1value "," arg2name ":

`" arg2value "}`. The field name must be identical to the variable name and the field value defines the variable value for the current execution of the ADIF export scheme.

> To specify the parameters `@lastupdate` and `@orga`, the JSON body of the request could for example be defined as:
>
> ```
> {
> "Scheme": "AppExport",
> "UserArgs": {"@lastupdate":"09/06/2018","@orga":"ITGroup"},
> "Verbose": false,
> "Synchron": true
> }
> ```

> If the specification of parameters is not correct in the call, the ADIF job is started, but execution might fail. Information about the success of the ADIF job execution is not returned to the RESTful service call. The return value from the RESTful service call will inform about the last successful execution step, which is start of the ADIF job. The success of the ADIF job execution can then be checked for example via the **ADIF Jobs Administration** functionality.
>
> For more information about the ADIF Jobs Administration functionality, see Executing and Controlling ADIF via the ADIF Jobs Administration and My ADIF Jobs functionalities in the Alfabet User Interface.

- **Starting ADIF via the ADIF Jobs Administration functionality in the Alfabet User Interface**

The ADIF jobs administration functionality does not provide a mechanism to set parameter values. If an ADIF job shall be started via the **ADIF Jobs Administration**, it should not contain parameters. If the ADIF scheme includes parameter definitions, it can only be started via the **ADIF Jobs Administration** functionality with the default values defined in the ADIF scheme for the parameters. The scheme must not include mandatory parameter definitions.

> For general information about starting ADIF via the **ADIF Jobs Administration** functionality, see Executing and Controlling ADIF via the ADIF Jobs Administration and My ADIF Jobs functionalities in the Alfabet User Interface.

- **Starting ADIF via an event**

If you start the ADIF job via an event, parameter values can be handed over during execution of the event via a query defined in the event template. In the attributes of the event template, the following attributes must be set:

- **Variable Names**: If the ADIF scheme contain parameter definitions, the names of the parameters must be written as comma separated list into this attribute. The specification must be case sensitive. The values for the parameters must then be provided with the attribute **Values for Variables via Query** / **Values for Variables via Query as Text**.

- **Values for Variables via Query** / **Values for Variables via Query as Text**: The query must return the values for the parameters defined in the attribute **Variable Names** in the same order. The query can be defined either via native SQL or via Alfabet query. Use the **Value for Variables via Query** attribute to define an Alfabet query. Use the **Values for Variables via Query as Text** attribute to define a native SQL query. Alfabet query language parameters can be used in the query. The base object returned via the Alfabet query language parameter

`BASE` depends on the way an event has been triggered. For events triggered via a workflow or wizard, `BASE` returns the REFSTR of the object the user was working on in the wizard step or workflow step triggering the event. For events triggered via an REST API call event, the `BASE` object is the same as the one for the event triggering the event.

The query must return a dataset with the column names identical to the parameter names defined with the attribute **Variable Names**.

For example if **Variable Names** is defined as `@AppName,@AppVersion`, the query might be defined as:

```
SELECT REFSTR, NAME AS'@AppName', VERSION AS'@AppVersion'
FROM APPLICATION
WHERE REFSTR = @BASE
```

For general information about starting ADIF via the Alfabet RESTful services, see the reference manual *Alfabet RESTful API*.

- **Starting ADIF via a button in the filter panel of a configured report or in the toolbar of an Alfabet view**

If an ADIF job is started via a button in the filter panel of a configured report or in the toolbar of an object view, the values for the parameter definition in the configured report can be set as follows:

- If the button is defined for an object view or a configured report that is assigned to a base object class via its **Apply to Class** attribute, the information about the base object the object view or configured report is opened for is returned as `@BASE`. If the ADIF scheme has a parameter definition with the **Name** set to `@BASE` and the **Parameter Type** set to `Reference`, this parameter will be substituted with the REFSTR of the current object on execution of the ADIF scheme.

- If the button is defined for a configured report that has filter fields, the values set in the filter fields are handed over as parameter values to the ADIF scheme when the ADIF job is started via the button. In the ADIF scheme, the **Name** of the parameter must be identical with the **Name** of the filter field and the **Parameter Type** must be identical to the data type returned by the filter field.

Please note that the filter field name must start with an @. Standard filter fields that are automatically generated by default in reports of the type `Query` based on an Alfabet query are starting with a colon instead of @ and must be adapted manually.

Comparison between filter fields and parameter names is case sensitive.

If the filters of a configured reports are not set when the ADIF job is executed, no value will be provided for the ADIF scheme parameter. Therefore, ADIF scheme parameters filled with values from filter fields that are not mandatory in the configured report should not be defined as mandatory in the ADIF scheme. Either a default value should be defined for the parameters or `NULL` value handling should be taken into account in the SQL commands of the ADIF scheme.

For general information about starting ADIF via a button, see Executing ADIF via a Button in the Alfabet User Interface.

Defining Parameter Values During Testing of the ADIF Scheme

When debugging an ADIF scheme with the ADIF Debugger, no command line can be defined. To set parameters for testing, the parameter values for testing can be written into the ADIF scheme.

To specify parameter values for testing in the ADIF scheme:

1) In the ADIF explorer, click the ADIF scheme that you want to specify parameter values for.

2) In the attribute window of the ADIF scheme, set the following attributes:

- **Debug Arguments**: Define the values for the parameters used in a comma-separated string. The format of the string must be <parameter name>=<value>. Please note that the values must not be written into single quotes. If single quotes are required in the query for the parameter data type, for example for strings, these will be set automatically when substituting the value in the query.

Mandatory parameters must be defined in the **Debug Arguments** to test the ADIF scheme via the ADIF Debugger. If the parameter is not mandatory and no value is defined for the parameter in the **Debug Arguments**, the ADIF Debugger will use the default value defined for the parameter. If no default value is defined, the parameter will be substituted with NULL.

For example, to specify export of applications for which the organization responsible for the application is variable and the last update of application data is variable, the **Debug Arguments** attribute can be specified as:

```
@lastupdate=09/10/2012,@orga=ITGroup
```

Configuring Logging Parameters

During export via the ADIF console application, log messages are written to:

- A temporary database table in the Alfabet database.

- The log file of the ADIF console application. By default, this is the log file `ADIF_Console.log` in the working directory of the ADIF console application.

Logging information is written to a log file in the following format:

```
<date and time> <message type> <message text>
```

For example:

```
2010-11-30T10:43:24.31Z DEBUG_INFO Query returns 314 records
```

Log messages are written to the log file in the language specified in the cultures of the Alfabet database. If the ADIF console application is run with a server alias, the culture specified as the default for the configuration of Alfabet will be used. If the ADIF console application is run with a remote alias, the culture of the remote alias configuration will be used.

For information about configuring cultures for the Alfabet database, see the section *Specifying the Cultures Relevant to Your Enterprise* in the chapter *Localization and Multi-Language Support for the Alfabet Interface*. in the reference manual *Configuring Alfabet with Alfabet Expand*.

For information about the culture of the remote alias, see the section *Configuration Attributes for the Alfabet Components* in the reference manual *System Administration*.

The timestamp is the UTC time (coordinated Universal Time) and may therefore differ from the time in your local time zone. The timestamp is written in ISO 8601 combined date and time format as year-month-day-Thour:minutes:secondsZ.

The message type can be one of the following:

- **ERROR**: An error occurred. The message describes the type of error.

- **WARNING**: Problems were encountered during the execution of the utility that are not as severe as an error. The process was executed but the result should be checked. The message describes the problem.

- **INFO**: Information about the normal execution of the utility is given.

- **DEBUGINFO**: Information about the import process is given. This information is also stored in the Alfabet database in a temporary database table.

> The debug info for execution of SQL commands returns the number of records that are involved when executing the SQL command. This number can be higher than the number of objects exported because of database triggers that are executed.

During the import process, emails containing the log information can be sent in a defined time interval to a configurable email address.

> The email is meant as an alive message of the system informing the recipient about the availability of an active ADIF process. The content of the email is limited to the one-line content that is written to the log file at the moment the email is generated and is therefore not suitable for debugging.

Configuring Log File Storage and Handling

The storage of log information and the sending of emails with process-related information is configurable either in the ADIF import scheme configuration or in the command line when starting the ADIF console application. If a configuration can be done both in the ADIF import scheme and the command line, the specification via the command line options will overwrite the configuration in the ADIF import scheme.

The following table gives an overview of the configuration options:

Configurable Logging Behavior	Attribute of the ADIF Scheme	Command line Option	Description
Time interval for sending emails containing current log information	Debug Heart Beat	`-heartbeat <time in minutes>`	Specify the time interval in minutes between sending emails with the current log file content of the ADIF console application during import. By default, the debug heart beat is set to -1. This means that no emails will be sent.

Configurable Logging Behavior	Attribute of the ADIF Scheme	Command line Option	Description
			NOTE: If the import process is finished before the time interval elapsed for the first time, no email will be sent.
email address of the recipient of the log information	Recipient Mail	`-recipient-mail` <email address>	Specify the email address of the person that shall receive the content of the log file of the ADIF console application via email in the configured time interval.
email address used as sender address in the emails with log information	Sender Mail	`-sendermail` <email address>	Specify the email address that shall be used as sender email address for emails sent via the ADIF console application.
Name of the log file of the ADIF console application	-	`-logfile` <filename>	Specify the name of the log file of the ADIF console application. Allowed file extensions are.log and.txt. The default name is `ADIF_Console.log`
Location of the log file of the ADIF console application	-	`-logpath` <path>	Specify the path to the log file of the ADIF console application. By default, the log file is stored in the working directory of the ADIF console application. The ADIF console application process must have write access rights to the specified directory.
Amount of data written to the log file	-	`-logverbose`	If set, all information about the process will be written to the log file. If not set, only error messages and information about the process start and end will be written to the log file.
Removal of old log messages from the log file	-	`-logclear` <number of days>	If `-nologappend` is set, a new log file will be created each time the utility is used with the same specification of `-logfile` and `-logpath`. The log file name will be extended with a timestamp specifying the current UTC time. **NOTE**: The scanning process can lead to drawbacks in performance.
Creation of a new log file for each process started	-	`-nologap-pend`	This option can only be used if `-nologappend` is not set. If `-nologappend` is set, the `-logclear` setting will be ignored. During logging, the log file will be scanned for log messages with a timestamp older than the number

Configurable Logging Behavior	Attribute of the ADIF Scheme	Command line Option	Description
			of days specified with `-logclear` and these messages will be deleted.
			If `-nologappend` is not set, logging information will be appended to the already existing log file each time the utility is used.
			NOTE : To restrict the file size, you can set the `-logclear` option to delete old log messages.

To set the parameters in the ADIF export scheme:

1) In the explorer of the ADIF configuration interface, click the ADIF export scheme that you want to configure. The attributes of the ADIF export scheme are displayed in the attribute window on the right.

2) In the attribute window, set the parameters listed in the table above as applicable.

Configuring Log Message Content

ADIF export schemes can be configured to include customer-defined log messages and data sets found via SQL commands in the log files during execution of the ADIF scheme. Log file content can be added to the ADIF export scheme via SQL command elements. SQL command elements for debug messages can be located in any **SQL Commands -** < **Action Type** > folder except the **SQL Commands - DataExport** folder.

During export execution, the SQL commands in the scheme are executed in the order specified by the command location in the XML definition. When the SQL command is due for execution, the message defined in the SQL command element and, if specified, the result of the SQL query as data set, are displayed in the log message.

In an ADIF export scheme that exports data about applications to a CSV file, information about all applications for which more than one version exists in the database shall be written to the log file prior to export execution. An SQL command of the **Result Type** `DebugDataSet` is added to the **SQL Commands - OnStart** folder of the export entry that triggers the export of the application data to the CSV file:
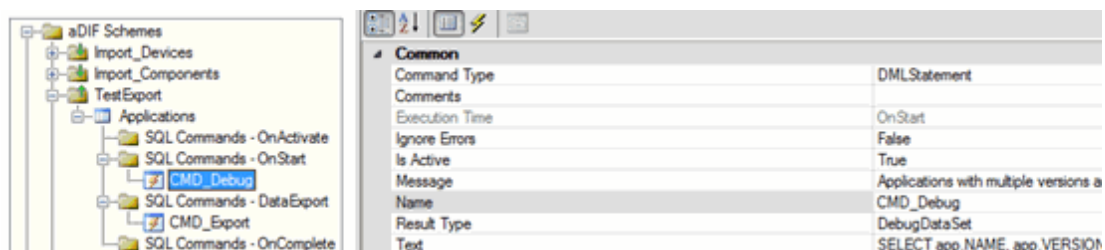
In the attribute window, the attribute **Result Type** is set to `DebugDataSet`. The data set to be displayed in the log file is defined by the SQL query defined in the attribute `Text`. Prior to the data set, a text explaining the data set shall be displayed in the log file. Therefore, an explanatory text is added in the attribute **Message**.

When executing the export, the **Message** text and the data set resulting from SQL query execution are added to the log file when the SQL command is due to be executed in the processing order of the ADIF scheme:

```
2011-01-03T13:57:54.937Z DEBUG_INFO  Scheme 'TestExport' has been initialized
2011-01-03T13:57:56.337Z DEBUG_INFO  Start process; Database: 'showcasePLM'. Driver: SqlServer. Commit after run = True.
2011-01-03T13:57:56.34Z DEBUG_INFO  Prepare database...
2011-01-03T13:57:56.367Z DEBUG_INFO  Process Entry 'Applications'
2011-01-03T13:57:56.37Z DEBUG_INFO  Get data set using CMD_Debug:
SELECT app.NAME, app.VERSION
FROM APPLICATION app
INNER JOIN APPLICATION app1
ON app.NAME =app1.NAME AND app.VERSION != app1.VERSION
WHERE app.NAME LIKE 'A%';
2011-01-03T13:57:56.377Z DEBUG_INFO  Applications with multiple versions are:
NAME|VERSION
ACCOUNT|1.2
ACCOUNT|1
AF Enterprise ConTrol|4.0
AF Enterprise ConTrol|3.1
AF Good Buy|3.0
AF Good Buy|2.0
AF HR Online|3.0
AF HR Online|2.2
2011-01-03T13:57:56.383Z DEBUG_INFO  Entry: Applications: Start CSV Export into 'TRIAL2.CSV'.
2011-01-03T13:57:56.383Z DEBUG_INFO  Execute Sql Command 'CMD_Export':
SELECT app.REFSTR, app.NAME, app.VERSION, app.CREATION_DATE AS "Creation Date", app.VARIANT
FROM APPLICATION app
WHERE app.NAME LIKE 'A%'
2011-01-03T13:57:56.393Z DEBUG_INFO  Query returns 22 records
2011-01-03T13:57:56.397Z DEBUG_INFO  Write into CSV File 'TRIAL2.CSV'
2011-01-03T13:57:56.4Z DEBUG_INFO  CSV File 'TRIAL2.CSV' has been closed.
2011-01-03T13:57:56.407Z DEBUG_INFO  Process has been finished.
```

(i) Debug messages and data sets can refer to the current import conditions via variables that depend on command line arguments of the ADIF console application when starting the export. For information on the use of variables to refer to the conditions defined in the command line, see Configuring Execution of the Import Scheme Dependent on Current Parameters.

Inserting a Static Text Message

To add a static text message to the log file:

1) In the explorer of the ADIF configuration interface, right-click the folder **SQL Commands - <Action Type >** that is executed in the order of commands when you want the text message to be displayed.

2) In the context menu, select **Create SQL Command**. A new SQL command element is added as child node of the **SQL Commands - <Action Type >** folder.

3) Click the new SQL command node and specify the following attributes in the attribute window:

   • **Name**: Enter a meaningful name for the SQL command.

   • **Comments**: Enter a comment that gives information about the functionality implemented by the SQL command.

   • **Result Type** : Select `DebugMessage`.

   • **Message**: Enter a text that shall be written to the log file at the position of the execution of this SQL command.

   • **Is Active** : Select `True` to activate the SQL command. Select `False` to deactivate the SQL command.

Inserting a Data Set

To include data from the database in the log file:

1) In the explorer of the ADIF configuration interface, right-click the folder **SQL Commands - <
Action Type** > that is executed in the order of commands when you want the text message to be
displayed.

2) In the context menu, select **Create SQL Command**. A new SQL command element is added as
child node of the **SQL Commands - < Action Type** > folder.

3) Click the new SQL command node and specify the following attributes in the attribute window:

- **Name**: Enter a meaningful name for the SQL command.

- **Comments**: Enter a comment that gives information about the functionality implemented by
the SQL command.

- **Result Type** : Select `DebugDataSet`.

- **Message**: Enter a text that shall be written to the log file at the position of the execution of
this SQL command prior to the data set resulting from the SQL query defined.

- **Text** : Define an SQL query with a `SELECT` statement that returns a data set. The resulting
data set is written to the log file at the position that indicates the execution of the SQL
command. If a text is also defined in the attribute **Message**, this text is written to the log file
prior to the data set. You can either write the SQL query directly into the attribute field or click
the **Browse** button to open an editor for the definition of the SQL query. Define the SQL
query in the **SQL Text** tab of the editor.

- **Message**: Enter a text that shall be written to the log file as an introduction to the data set.

- **Ignore Errors** : Select `True` if you want the export to be executed even if the SQL statement
specified with the attribute **Text** results in an exception. Select `False` if you do not want the
import to be executed in case of an error in the SQL statement.

- **Is Active** : Select `True` to activate the SQL command. Select `False` to deactivate the SQL
command.

Chapter 6:   Debugging an ADIF Configuration

The ADIF configuration user interface includes a debugging tool. The debugging interface allows data import or export to be run as a test with no persistent impact to the Alfabet database so that you can control the success of command execution step by step. You can access the debugging tool via the context menu of the ADIF import or export schemes.

> ⚠ For security reasons, ADIF configuration and debugging should be performed in a test environment to ensure that no damage occurs to the database in the production environment in case the configuration of the ADIF scheme or the import data used for import contains errors.

Understanding the Debugger Interface

When the debugger is started for a selected ADIF scheme, a new window opens that displays the structure of the selected ADIF scheme in an explorer. To the right of the explorer, a debug window opens. Log messages and information about the process step that is currently being performed are displayed in this window.

> ⓘ In addition to displaying the log information in the user interface, the ADIF debugger stores the log messages in a log file in the directory $APPDATA/Temp/Alfabet/<server name>/Runtime/temp.
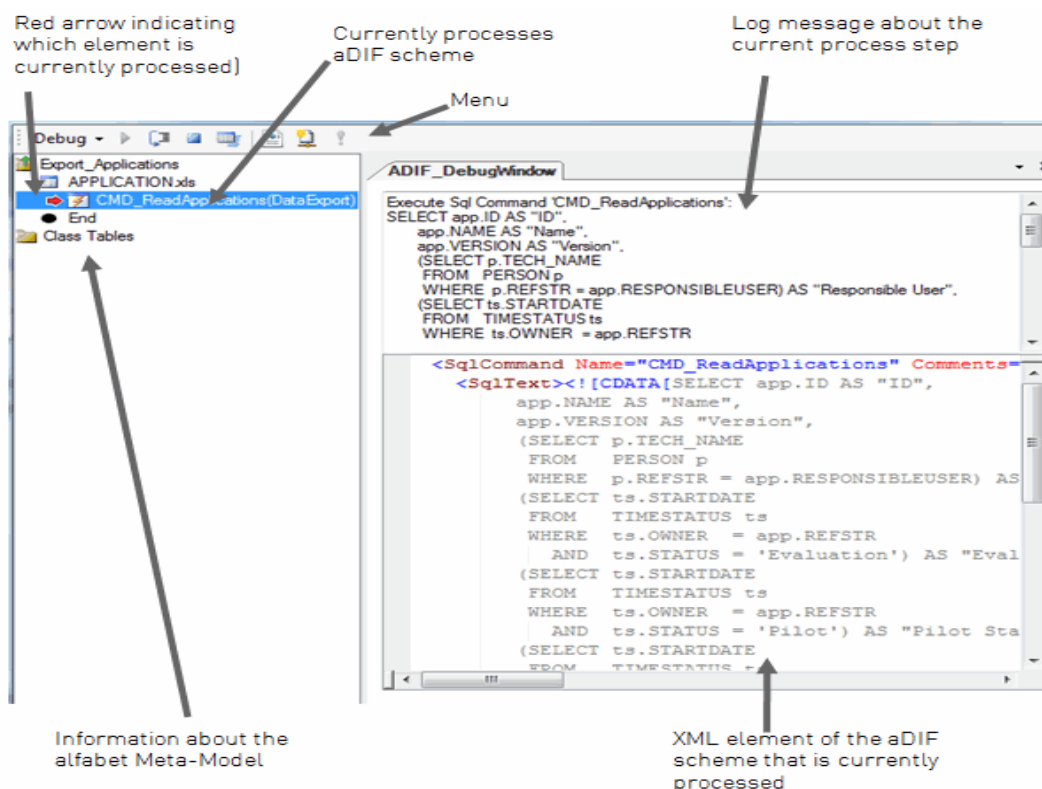
FIGURE: Interface of the ADIF Debugger

A menu is available at the top of the explorer that allows you to control the debugging process.

A **Class Tables** node is displayed in the explorer below the ADIF scheme node. All relevant database tables in the Alfabet database are listed under the **Class Tables** node. You can search the tables via SQL queries or display information about the meta-model structure that applies to the database table. If the ADIF scheme includes the definition of temporary tables, these tables are also listed in the explorer under a separate node **Temporary Tables**. For details about the information available via the database table nodes, see *Performing a Debug Run*.

Understanding the Debugging Process

The following workflow is required for debugging:

- If your database is configured in an XML file, upload it to the database. (optional)

- Configure your ADIF scheme to roll back import/export after the test run.

- Start the debugger.

- Start the debug process and go through the process step-by-step while controlling the results via the log messages and the resulting changes in the database tables.

- Correct any errors in the configurations, if necessary, and re-start the debugging process until the ADIF scheme results in the desired data manipulation

- Stop the debugger.

- Configure the ADIF scheme to write data persistently to the database.

- Export your ADIF scheme if you want it to be available via an XML file. (optional)

The following information is available:

- [Required Configuration of the ADIF Scheme Prior to Debugging](#)

- [Performing a Debug Run](#)

- [Required Configuration of the ADIF Scheme After Debugging](#)

Required Configuration of the ADIF Scheme Prior to Debugging

Debugging can only be performed on ADIF schemes that are uploaded to the Alfabet database. If you defined your ADIF scheme via an external XML editor, you must import it to the Alfabet database via the ADIF configuration interface. For more information about the import and export of ADIF schemes, see the section *External Editors* in the chapter *Configuring ADIF Schemes*.

The debugging process performs all steps configured in an ADIF scheme in order to help you to find and correct configuration steps that lead to errors or undesired data processing. The debug process is like a test run and therefore should not perform any changes to existing data in the Alfabet database or the external database that is targeted for data export. Therefore, the ADIF scheme must be configured to roll back all steps performed during testing:

1) In the ADIF explorer, click the ADIF scheme that you want to debug.

2) In the attribute window of the ADIF scheme, set **Commit After Run** to `False`.

⚠ The following restrictions apply to setting **Commit After Run** to `False`:

- **Commit After Run** only affects database transactions. If you export data to a file, the export file is created and data is added to the file also if **Commit After Run** is set to `False`.

- Setting **Commit After Run** to `False` rolls back all changes caused by DML statements (changes to data records in existing tables). Creating or deleting tables is not included in the roll back. This means, for example, if you test an ADIF scheme that is configured to write temporary tables to the database persistently, these temporary tables will be created persistently even if **Commit After Run** is set to `False`. SQL Commands of the type `OnActivate` are also excluded from roll back.

- When new objects are created during an import job, the data bind mechanism assigns `REFSTR` values for the new objects. When **Commit After Run** is set to `False`, the objects are not created in the database, but nevertheless the `REFSTR` values are regarded as in use and will not be used for data bind in the next ADIF run unless the Alfabet Server or Alfabet Expand application used to process the ADIF jobs is restarted.

- Changes triggered by `OnActivate` commands are not rolled back if the option **Commit After Run** is set to `False` for the import scheme.

It is possible to configure ADIF schemes to include variables in SQL queries. The values of the variables are provided in the command line of the ADIF console application during execution of the import or export. Default values for the variables can be defined in the ADIF scheme. During the import or export process, the values for the variables are read from the command line or, if not available from the command line, from the ADIF scheme configuration and written to a temporary database table for use during import. The name of the temporary database table is configurable.

ⓘ For information about defining parameters, see Configuring Import Dependent on Parameters or Configuring Import Dependent on Parameters.

The debugger does not allow variable values to be set during the debug run. Therefore, the variables must be defined in the ADIF scheme:

1) In the ADIF explorer, click the ADIF scheme that you want to debug.

2) In the attribute window of the ADIF scheme, set the following attributes:

- **Debug Arguments**: Define the values for the variables used in a comma-separated string. The format of the string must be <variable name>=<value>.

  ⓘ Note the following when specifying variables:

  - Mandatory parameters must be defined in the **Debug Arguments** to test the ADIF scheme via the ADIF Debugger. If the parameter is not mandatory and no value is defined for the parameter in the **Debug Arguments**, the ADIF Debugger will use the default value defined for the parameter. If no default value is defined, the parameter will be substituted with `NULL`.

  - The values must not be written into single quotes. If single quotes are required in the query for the parameter data type, for example for strings, these will be set automatically when substituting the value in the query.

- The specification of both variable and value name is case-sensitive.

- **Arguments Table Name** : Optionally, specify a name for the temporary database table for the storage of the variable names and values during the import / export process. The default table name is `ADIF_ARGS`.

Performing a Debug Run

Debugging is performed in the debugger window of the ADIF configuration interface.

ⓘ A debug run is like a test export or import of data. If you want to test the import of data, you must provide data to be imported on the local file system in a ZIP file or enable access to the database that you want to import data from.

1) In the ADIF explorer, right-click the ADIF scheme that you want to debug and select **Start Debugger**.
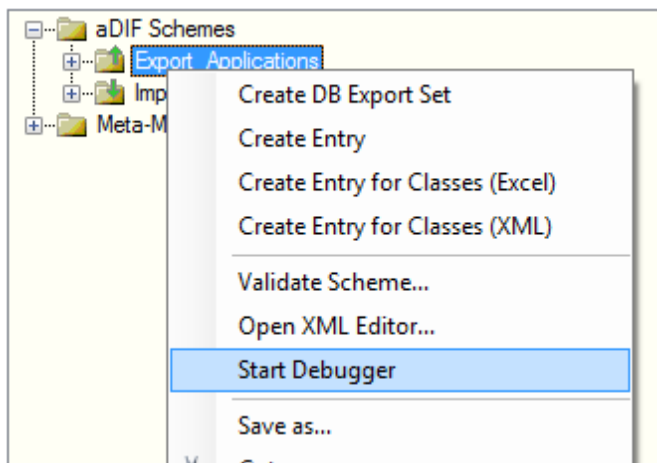
The debugger opens with the selected ADIF scheme displayed in the explorer.

2) In the menu, click the **Start** ▶ button.

3) If import from files or export to files is configured in the ADIF scheme, an explorer window will open.

- For export schemes, specify the name and location of the.zip file where data shall be exported. If the file does not exist, it will be created during the process. If it does exist, it will be overwritten.

- For import schemes, specify the name and location of the.zip file containing the files where the data that shall be imported.

In the explorer, a red arrow is displayed in front of the ADIF scheme indicating the starting point for the import/export process. A Debug Window opens that provides information that the ADIF scheme was initialized and displays the XML code of the currently processed element of the ADIF scheme.
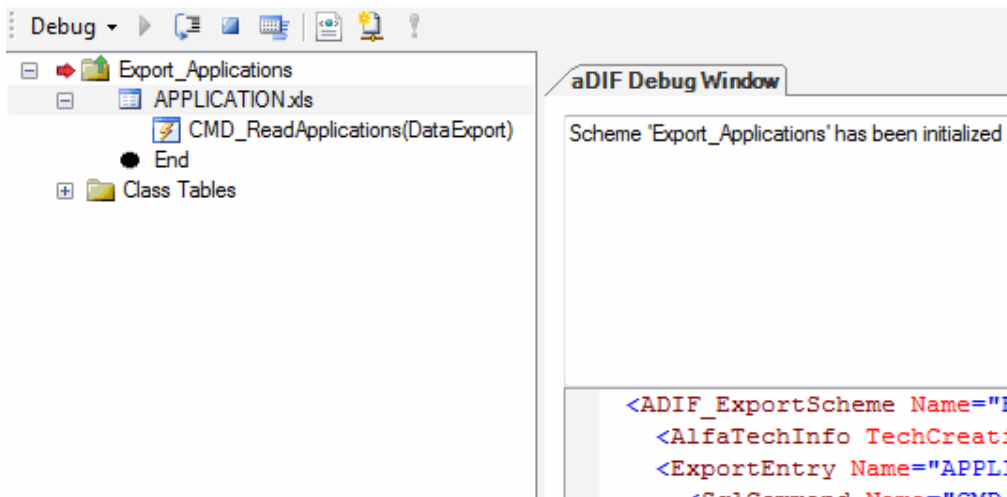
4) In the menu, click the **Step Over** ⌈⃗ button. The next step of the import or export process is performed. The message in the **ADIF Debug Window** changes and provides information about the success of the next step.

5) Repeat step 4 for each process step. You can use the following mechanisms to check whether the result of your configuration is as expected:

- Check the error message in the **ADIF Debug Window** to be notified about the success of the step execution. If an error occurs, the red arrow in the explorer indicates which part of the import scheme is affected. The XML code for the element is displayed in the debug message so that you can scan the code for errors.

- To check the XML code of any other element of the ADIF scheme, click the element in the explorer and click the **Show Definition** button in the menu of the ADIF debugger.
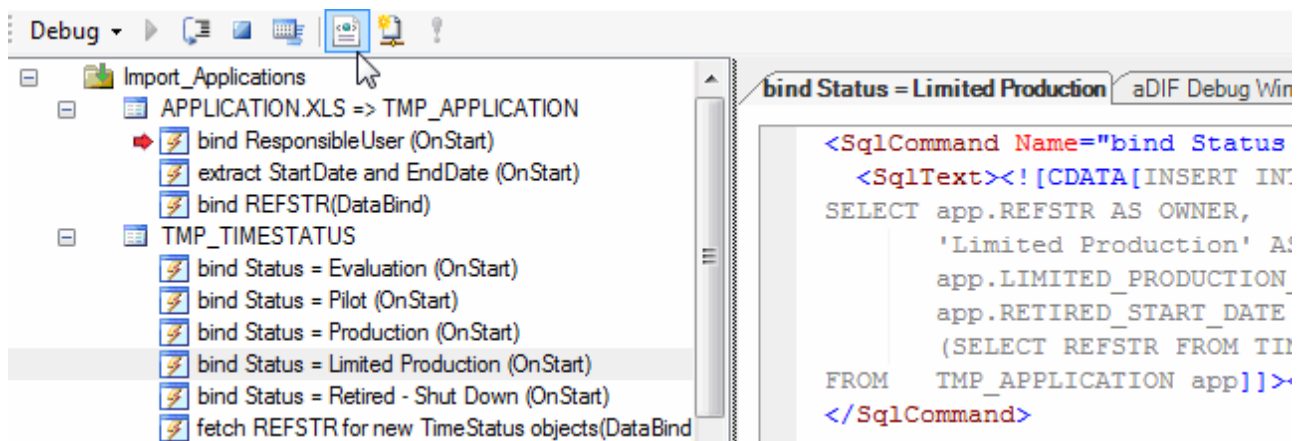
- To check whether unexpected results are caused by wrong interpretation of the Alfabet meta-model, information can be displayed about the object classes of the Alfabet meta-model and their properties. To see a documentation of the requirements for data import to the database table of an object class, right-click the class under the **Class Tables** sub-node of the ADIF debugger's explorer and select **Show Class Definition**.
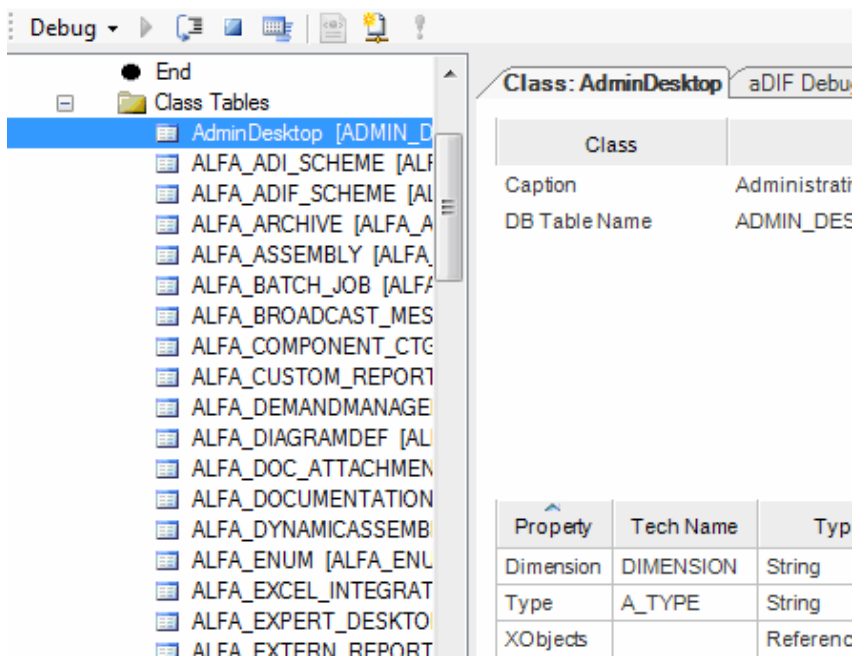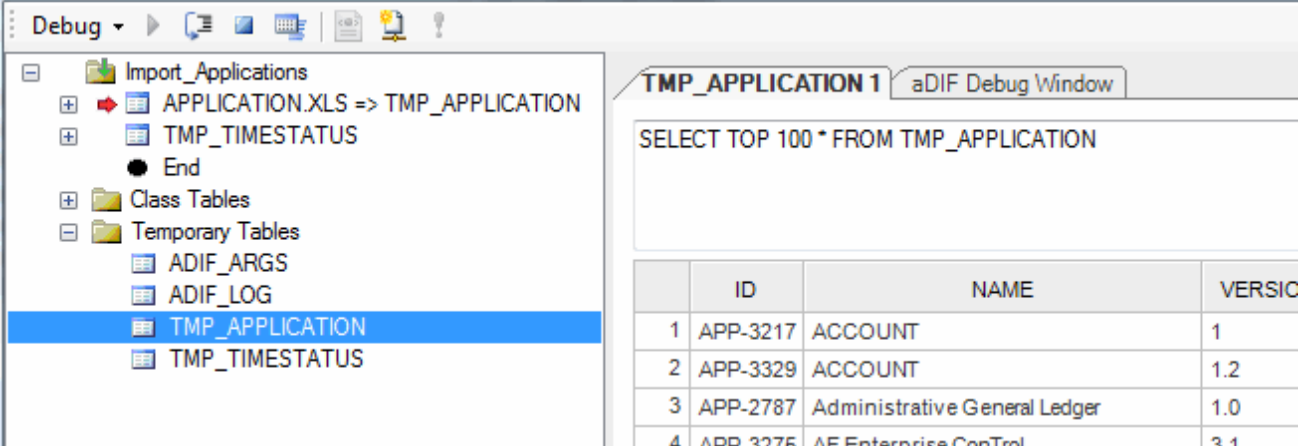
- When the result of a step is the creation of a temporary table, you can see the temporary table listed under the sub-node **Temporary Tables** of the ADIF debugger's explorer.
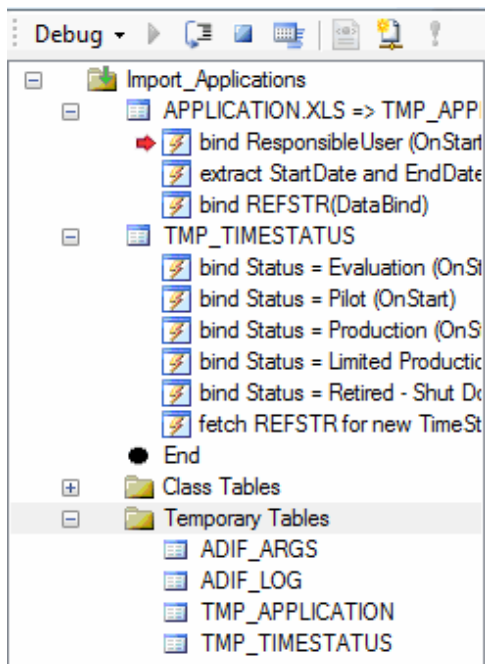
- When data is written to either a temporary table or a database table of the Alfabet meta-model, check the resulting data sets in the table. To check the content of a database table, click the database table under either the **Class Tables** or **Temporary Tables** sub-node of the explorer, and click the **New SQL** button in the menu. A new window opens with a predefined query that finds the first hundred rows in the database table. You can alter that query to search the database table for the expected changes. In the menu, click the **Execute SQL** button to see the result of your query.

ⓘ The `RELATIONS` table is not displayed as sub-node of the **Class Tables** node. Never-theless, it can be controlled by clicking the **New SQL** 🔖 button and defining a SE-LECT statement targeting the `RELATIONS` table.

ⓘ It is recommended that you check the changes to database tables even if a step creating table entries does not lead to an error message. Error messages in a later step can be caused by missing, incomplete, or wrong data generation in a previous step.

ⓘ If the Alfabet database is located on an Oracle® database server, integers are dis-played as numbers of the data type Real. Oracle® database servers store integers in the data type NUMBER and returns decimal numbers as an SQL search result.

6) In case of an error in the progress of data import or export, click the **Stop** 🔲 button in the menu, close the debugger and correct the error in the ADIF scheme configuration via the ADIF explorer. Then proceed with step 1 of the debugging process.

ⓘ A rollback of data is performed when closing the debugger independent of the attribute **Commit After Run**.

7) After the successful execution of the debugger, a log file will be displayed that provides information about the overall progress of the process. Exit the debugger.

Required Configuration of the ADIF Scheme After Debugging

After successful debugging, the ADIF scheme must be configured to write data persistently to the Alfabet database:

1) In the ADIF explorer, click the ADIF scheme that has been debugged.

2) In the attribute window of the ADIF scheme, set **Commit After Run** to True.

If you want the ADIF scheme to be available as an XML file, you must export it from the Alfabet database via the ADIF configuration interface. For more information about the import and export of ADIF schemes, see the section *External Editors* in the chapter *Configuring ADIF Schemes*.

Chapter 7: Starting Data Import, Export, or Manipulation via ADIF

After you have configured the ADIF interface and controlled your configuration via the ADIF debugger, you can use the configured ADIF scheme for data import or export or data manipulation either on request or on a regular basis.

There are a number of options to execute ADIF import, export or manipulation of data:

- Software AG provides a Windows® command line tool `ADIF_Console.exe` to execute import, export or manipulation of data.

- The Alfabet RESTful service provide the ability to execute ADIF import or export schemes via a RESTful service call.

- The ADIF import or export can be triggered via an event that is started in the background when a user enters or exits a wizard or workflow step.

- ADIF import or export can be triggered by the user via a customer defined button added to the toolbar of an Alfabet view on the Alfabet user interface.

- ADIF import or export can be controlled or triggered from the Alfabet user interface in the functionality **ADIF Jobs Administration**.

- ADIF import or export can be scheduled for future execution via the **Job Schedule** functionality.

ⓘ ADIF imports may trigger a high number of single database insert and delete transactions. This can have a negative impact on fragmentation of indices. As a result, CPU usage is increased and performance is reduced. It is recommended to rebuilt indexes in regular intervals for object classes subject to ADIF import. This can be done using the command line tool `AlfaAdministrator-Console.exe`. For more information, see *Rebuilding Indexes on Database Tables* in the reference manual *System Administration*.

The following information is available:

- Executing ADIF via a Command Line Tool

- Executing ADIF via RESTful Service Calls

- Executing ADIF via an Event

- Executing ADIF via a Button in the Alfabet User Interface

    - Configuring the ADIF Scheme to be Executable via the User Interface

    - Adding a Button for ADIF Execution to a Configured Report

    - Adding a Button for ADIF Execution to an Object View

- Executing and Controlling ADIF via the ADIF Jobs Administration and My ADIF Jobs functionalities in the Alfabet User Interface

    - Configuring Availability of ADIF Execution and Control in the User Interface

        - Configuring Asynchronous Execution

    - Controlling Success of Executed ADIF Jobs

    - Executing ADIF

Executing ADIF via a Command Line Tool

After you have configured the ADIF interface and controlled your configuration via the ADIF debugger, you can use the configured ADIF scheme for data import or export or data manipulation either on request or on a regular basis.

Software AG provides a Windows® command line tool `ADIF_Console.exe` to execute import, export or manipulation of data.

⚠️ For security reasons, the following recommendations should be considered for the import of data to the Alfabet database or for export of data to external databases:

- Test the import of data in a test environment prior to the import to a production environment for each import performed via the ADIF console application to ensure that the correct changes are applied.

- Backup the production database prior to the import of data.

Executable	`ADIF_Console.exe` located in the **Programs** sub-directory of the installation directory of the Alfabet components.
Preconditions	An `AlfabetMS.xml` configuration file with a server alias configuration specifying the connection to the Alfabet database is available and the respective functionality is configured.

Logging	Log information is written to a log file and to the Alfabet database. Logging is configurable via the ADIF scheme used or in the command line. For more information about logging, see *Configuring Logging Parameters* and the list of command line options below.
Command line help	Start executable with `-h` or `-help`

Executables for batch jobs can be started in a command line or by means of a Windows® batch job.

When executing a Windows batch job, the execution time for a batch job is defined with the help of the Windows scheduler for batch jobs. When starting the executable with a command line, the batch job is executed immediately.

The setting in the **Application Server** tab must be identical in all server alias configurations of all Alfabet components including batch utilities.

Providing Relevant Data and Configuration

The ADIF console application requires access to the following files to be executed:

A fully configured ADIF scheme

The ADIF import or export scheme must be completely configured and debugged before starting the data import or export. The ADIF scheme can be provided as an XML file on the local host or as part of the Alfabet database.

> ⚠️ When using an ADIF scheme from an XML file, you must ensure that no ADIF scheme with a **Name** attribute exists in the Alfabet database that is identical to the **Name** attribute of the ADIF scheme in the XML file.
>
> The ADIF scheme in the XML file is temporarily loaded to the Alfabet database during import and deleted from the Alfabet database after the import/export process. If an ADIF scheme with the same **Name** is available in the Alfabet database when starting the ADIF console application, this ADIF scheme will be overwritten by the ADIF scheme in the XML file and deleted after the import/export process.

An AlfabetMS.xml configuration file

For access to the Alfabet database, an `AlfabetMS.xml` configuration file with an alias configuration must be available.

The way batch jobs are executed depends on the setting of the attributes in the Application Server tab of the server alias configurations of the Alfabet components:

- If **Use Event Queue for All Jobs** is selected, all server processes are scheduled via event queueing. The ADIF console application schedules a job for execution of the ADIF job in the Alfabet database. The Alfabet Server scans the queue of jobs to be processed in regular intervals and processes the incoming jobs. More than one ADIF job can be processed in parallel via different threads. The Alfabet Server only starts jobs in parallel if they do not write data into the same database table in the Alfabet database.

This is the recommended option that will be maintained in future releases.

- If **Use Application Server and Net Remoting Service** is selected, the ADIF console application hands the job over to the Alfabet Server for execution. The ADIF console application waits until the Alfabet Server finishes the execution and then retrieves the log and the success status from the Alfabet Server. The Alfabet Server executes only one ADIF job at the same time. By default, if you try to start an ADIF job while another job is running, your new job will be terminated with an error message. However, you can configure the ADIF console to queue jobs for execution. A maximum queue time must be defined in the command line to queue the job. If a queue time is defined, and a job is started while another job is still running, the new job will be queued and will start when the first job is finished. If the job is queued longer than the defined maximum queue time, it is deleted from the queue without being executed.

If a remote alias is used for execution of the ADIF console application, the corresponding Alfabet Server must be running. If a server alias is used, all other Alfabet applications with access to the Alfabet database must be stopped.

This method will be deprecated in the future because it will not be compatible with the upcoming .NET Core component.

When starting the ADIF console application, the alias configuration name must be specified in the command line. If the `AlfabetMS.xml` configuration file containing the alias configuration is not located in the working directory of the ADIF console application, the path to the file must additionally be specified in the command line.

For import: Access to data that shall be imported

If the data shall be imported from files, the files containing the data must be located either in a single *.zip file or in a single import directory. If files are located in an import directory, the ADIF console application will create a ZIP file containing all files located in the import directory for use in data import as part of the import process.

⚠️ When import shall be performed from an XML file referring to a DTD file, the DTD must also be located in the import ZIP file or import directory respectively.

Files are uploaded to the Alfabet database and deleted from the database after import was performed. Whether the import files on the local host are changed by the import process depends on the import scheme configuration. For more information, see *Configuring Data Import with ADIF*.

If data is imported from multiple import files in one ADIF import and a specified import files are missing, import from all other files is performed and a warning is written to the log file that import of data from the missing file could not be performed.

Starting the ADIF Console Application

Start `ADIF_Console.exe` with the command line options listed in the table below to perform a data import based on an ADIF import scheme. If a command line option requires a parameter definition and the parameter includes whitespaces, the parameter must be written in inverted commas. For example:

```
ADIF_Console.exe -import -msalias Alfabet -scheme "Application Import"
```

For the import of data, you must start `ADIF_Console.exe` with at least:

- a specification of the alias configuration defining access to an Alfabet Server or a Alfabet database,

- the import scheme and, if data is imported from files,

- the location of the import files in the local file system:

```
ADIF_Console.exe -import -msalias <alias name> [-scheme <scheme
name> -importfile <file name>.zip]
```

For export of data, you must start `ADIF_Console.exe` with at least:

- a specification of the alias configuration defining access to an Alfabet Server or a Alfabet database,

- the export scheme and, if data is exported to files,

- the location in the local file system where the export files shall be stored

```
ADIF_Console.exe -export -msalias <alias name> -alfaLoginName <user
name> -alfaLoginPassword <user password> [-scheme <scheme name> -
importfile <file name>.zip]
```

Command Line Option	Mandatory/ Default	Explanation

-<action>	Mandatory	To import data, start the console application with
		
		    -import
		
		To export data, start the console application with
		
		    -export
		
		In the command line help that can be called with the -h command line option, a -update action type is also listed next to -import and -export. This option is for internal use by Software AG only and must not be used.

-msalias <alias name>	Mandatory	Enter the alias name as specified in the AlfabetMS.xml configuration file.
-msaliasesfile < Alfabet configuration file path>	Optional	If the AlfabetMS.xml configuration file that contains the specification of the alias is not located in the same directory as the executable, the path to the AlfabetMS.xml file must be specified with this parameter.
-alfaLoginName <user name>	Mandatory	User name for access to the Alfabet database. The user specified by this login name will be written to the audit history of changed objects as responsible for changing the data. ⚠ A user can only execute a batch job if the **Can Execute Batch Jobs** checkbox is selected (=True) for the user.

Command Line Option	Mandatory/ Default	Explanation
`-alfaLoginPassword <user password>`	Optional	Password for access to the Alfabet database.
`-scheme`	Optional	Specify the name of the ADIF scheme stored in the Alfabet database that contains the configuration of the data import/export. **NOTE** : It is mandatory to specify this option or the option `-schemefile`. Only one of the options can be specified in a command line. Which of the options is applicable will depend on the location of the ADIF scheme.
`-schemefile`	Optional	Specify the name of and path to the XML file of the ADIF scheme that contains the configuration of the data import/export. **NOTE** : It is mandatory to specify this option or the option `-scheme`. Only one of the options can be specified in a command line. Which of the options is applicable depends on the location of the ADIF import scheme. ⚠ If a scheme with the same name as the scheme in the external XML file exists in the Alfabet database when running the job, the scheme in the Alfabet database will be deleted. This mechanism is implemented to ensure that only one version of a scheme exists. If the scheme is called via an external XML file, it is supposed that the scheme with the same name in the Alfabet database is outdated and no longer used.
`-waittime`	Optional	Only applicable if Alfabet components are using event queueing for processing of ADIF jobs. Specify the time in minutes that the process shall wait if another ADIF process is running. Two processes cannot run in parallel. A new process is only queued for execution if the command line option `-waittime` is set. If the currently running process is not finished during the configured wait time, the queued process will be deleted. If `-waittime` is not set and a process is started while another process is currently running, the new process is terminated and an error message is given.
`-idletime`	Optional	Only applicable if Alfabet components are using event queueing for processing of ADIF jobs. The batch utility waits for alive messages from the Alfabet Server and if none are received within 60 seconds, the job will be cancelled. The allowed wait time for alive messages can be changed by defining a new time interval in seconds with `-idletime`.

Command Line Op-tion	Mandatory/ Default	Explanation
-synchronously	Optional	Only applicable if the Alfabet components are using event queue-ing for processing of ADIF jobs. If this parameter is added to the command line, the ADIF console application will check the event queue for the status of the scheduled event until the event is processed and will return the status of the finished ADIF job exe-cution.

-<variable name>	Optional	When the ADIF scheme is configured to use variables, the varia-bles can be specified in the command line. For each variable, a separate command line option must be specified as
		
		    `-<variable name> <value>`
		
		For example:
		
		    `-@fiscalyear 2009`
		
		For more information about the use of variables, see *Configuring Execution Dependent on Current Parameters* in the description of the export configuration or *Configuring Import Dependent on Pa-rameters* in the description of the import configuration.

Import-specific command line options

-importfile	Optional	Specify the name of and path to the ZIP file containing the files from which data shall be imported. **NOTE** : If the import scheme configuration defines import from files, it is mandatory to specify this option or the option `-im-portdir`. Only one of the options can be specified in a command line.
-idletime	Optional	Specify the name of and path to the directory containing the files from which data shall be imported. **NOTE** : If the import scheme configuration defines import from files, it is mandatory to specify this option or the option `-import-file`. Only one of the options can be specified in a command line.

Export-specific command line options

-exportfile	Optional	Specify the name of and path to the ZIP file to which the data will be written. If the file does not exist, it will be created; if it does ex-ist, it will be overwritten. If the path does not exist, the ADIF ex-port cannot be executed. The export mechanisms can create files, but it cannot create directories.

Command Line Option	Mandatory/ Default	Explanation
		NOTE: If the ADIF export scheme configuration defines export to files, it is mandatory to specify this option.
`-unzip <true|false>`	Optional	If set to `true`, the ZIP file to which the export data is written is unzipped after the export. The files with the export data are then located in the directory specified as the location for the ZIP file as well as in the ZIP file. If set to `false`, the ZIP file to which the export data is written is not unzipped and the data is available in zipped format only.

Logging-specific command line options

Command Line Option	Mandatory/ Default	Explanation
`-logpath` <log file path>	Log file is stored in the working directory of the executable.	Specification of a path to a directory for storage of the log file.
`-logfile` <log file name>	<Executable>.log	Specification of the log file name. Allowed file extensions are LOG and TXT.
`-logverbose`		If `-logverbose` is set, additional information about the running process is logged. The content of additional information messages depend on the utility used. **NOTE**: Verbose logging is in most cases not required and can lead to a decrease in performance.
`-nologappend`		If `-nologappend` is set, a new log file is created each time the utility is used with the same specification of `-logfile` and `-logpath`. The log file name is extended with a timestamp specifying the current UTC time. If `-nologappend` is not set, logging information is appended to the already existing log file each time the utility is used. **NOTE** : To restrict the file size, you can set the `-logclear` option to delete old log messages.
`-logclear` <number of days>	Infinite	This option can only be used if `-nologappend` is not set. If `-nologappend` is set, the `-logclear` setting is ignored. During logging, the log file is scanned for log messages with a timestamp older than the number of days specified with `-logclear` and these messages are deleted.

Command Line Option	Mandatory/ Default	Explanation
		NOTE: The scanning process can lead to drawbacks in performance.
`-heartbeat`	-1	Specification of the time between sending emails with the current content of the log file to the recipient specified with the option `-recipientmail`. If -1 is specified or a heartbeat is not defined, no email will be sent out and log information is exclusively available via the log file of the ADIF console application.
		NOTE: The email is meant as an alive message of the system informing the recipient about the availability of an active ADIF process. The content of the email is limited to the one-line content that is written to the log file at the moment the email is generated and is therefore not suitable for debugging.
		NOTE: The interval for sending the emails can also be specified with the **Debug Heart Beat** attribute of the ADIF scheme specified with the option `-scheme` or `-schemefile`. A specification in the command line overwrites a specification in the ADIF scheme.
		NOTE : The recipient's and sender's email address must also be specified either in the command line or in the ADIF scheme specified with the option `-scheme` or `-schemefile` to send information about the logging process via email.
`-sendermail`		Specification of the email address used as sender address in emails with log information sent out in the interval specified with `-heartbeat` to the recipient specified with `-recipientmail`.
		NOTE: The sender email address can also be specified with the **Sender Mail** attribute of the ADIF scheme specified with the option `-scheme` or `-schemefile`. A specification in the command line overwrites a specification in the ADIF scheme.
`-recipientmail`		Specification of the email address used as recipient address in emails with log information sent out in the interval specified with `-heartbeat`.
		NOTE: The recipient email address can also be specified with the **Recipient Mail** attribute of the ADIF scheme specified with the option `-scheme` or `-schemefile`. A specification in the command line overwrites a specification in the ADIF scheme.

Executing ADIF via RESTful Service Calls

ADIF export or import can be triggered via a RESTful service call to the RESTful API of the Alfabet Web Application.

The service call requires that the Alfabet RESTful service interface is activated and all access permissions are set accordingly.

The execution of ADIF import or ADIF export via a RESTful service call can either be triggered by an external application sending the request to the RESTful API of the Alfabet Web Application or via an event that is automatically triggered from a workflow step or a wizard step.

For information about the required settings and the service call for execution ADIF import or ADIF export from a RESTful service client, see the reference manual *Alfabet RESTful API*.

For information about the configuration of events, see *Configuring Events* in the reference manual *Configuring Alfabet with Alfabet Expand*.

Executing ADIF via an Event

Event management provides a means to automatically execute an ADIF import or export job automatically as a consequence of a defined user action on the Alfabet user interface. The ADIF import or export can be executed either on the same or on a different Alfabet database.

The following user activities can be specified to trigger events:

- When a wizard step is entered, exited, or cancelled.

- When a workflow step is entered, refused, expired, or exited.

- When an event template configured to send a RESTful service call is finished.

The execution of the event is performed via a RESTful service call to the RESTful API of the Alfabet Web Application that triggers the ADIF scheme execution as defined in the event.

For information about the configuration of events, see *Configuring Events* in the reference manual *Configuring Alfabet with Alfabet Expand*.

Executing ADIF via a Button in the Alfabet User Interface

You can start ADIF jobs via a custom button in the filter panel of a configured report or in the toolbar of custom object views. The toolbar button is then available for the object profile and all object cockpits defined for the object view.

The ADIF scheme triggered via the ADIF job can include parameters that refer to the filter fields in the filter panel of the configured report as well as to the base object of the configured report or the object that the user is working with in the object view. For information about how to include parameters into ADIF schemes, see Configuring Execution of the Import Scheme Dependent on Current Parameters in the chapter Configuring Data Export with ADIF or Configuring Import Dependent on Parameters in the chapter Configuring Data Import with ADIF.

If the ADIF job exports data to a file, the exported file or files will be provided for download in a zip file named <ADIF scheme name>_<timestamp>.zip.

If the ADIF job imports data from file, the user is prompted to select a file from the local file system.

Success of the execution can be controlled via the functionalities ADIF Jobs Administration and My ADIF Jobs. For more information, see Executing and Controlling ADIF via the ADIF Jobs Administration and My ADIF Jobs functionalities in the Alfabet User Interface.

To start an ADIF job via a button in a configured report or object view, both the ADIF scheme the job is executing and the configured report or object view must be configured as described in the following sections:

- Configuring the ADIF Scheme to be Executable via the User Interface

- Adding a Button for ADIF Execution to a Configured Report

- Adding a Button for ADIF Execution to an Object View

Configuring the ADIF Scheme to be Executable via the User Interface

To configure an ADIF scheme to be executable via the Alfabet user interface:

1) In the explorer of the **ADIF** tab of Alfabet Expand, click the ADIF scheme that shall be executed via button interaction.

2) In the attribute window, set the attribute **Alfabet User Interface Behavior** to `VisibleExecutable`.

Adding a Button for ADIF Execution to a Configured Report

To create a button for ADIF execution in the filter panel of an existing configured report with a filter panel:

1) In the explorer of the **Reports** tab of Alfabet Expand, expand the node of the configured report that you want to add a button to.

2) Double click on the report view node beneath the configured report node. The report view opens in the middle pane.

3) In the toolbar of the middle pane, click the **Button** ![ab] interface control and then click into the filter panel. A new button is added to the filter panel. You can optionally change the location and size of the field on demand.

4) In the attribute window, define the following attributes for the new button:

- **Name**: Optionally, change the technical name of the button to a meaningful value. The name is a technical name and should not contain special characters or whitespaces.

- **Caption**: Change the caption of the button to the text that shall be displayed in the button to the user on the Alfabet user interface.

- **Hint**: Enter text for a tooltip. The tooltip will be displayed when the user points to the button.

- **Sub-Type** : Select `ADIF_Button`.

- **XML Definition**: Click the **Browse** ![...] button and enter the following XML code into the editor that opens:

```
<ADIF_Button Type="ExportType" Scheme="SchemenameFileRequired="false">
</ADIF_Button>
```

Set the Type XML attribute to Export to trigger an ADIF export or to Import to trigger an ADIF import.

Enter the name of the ADIF scheme that shall be executed into the Scheme XML attribute.

If the ADIF scheme shall not export to a file, add the XML attribute `FileRequired` and set it to `false`.

5) In the toolbar of Alfabet Expand, click the **Save** ![save icon] button.

Adding a Button for ADIF Execution to an Object View

To create a button for ADIF execution in the toolbar of an existing custom object view:

1) In the explorer of the **Presentations** tab of Alfabet Expand, navigate to the custom object view that you want to add a button to.

2) Right-click the custom object view ![icon] and select **Create Button**. The new button ![icon] is displayed below the custom object view node.

3) Click the new button to activate the attribute window and define the following attributes, as needed:

   - **Button Type** : Select `Action`.

   - **Name**: Enter a unique name for the button. Special characters are not allowed. This value is displayed in the solution interface if the **Caption** attribute has not been defined.

   - **Caption**: Enter the caption that should be displayed on the button. If no caption is defined, the value in the **Name** attribute will be displayed on the button.

   - **Operation** : Select `ADIF_Operation`.

   - **Apply Operation To** : If the ADIF scheme uses the parameter `@BASE` to refer to the current object, set this parameter to `Base Instance` to define that the button will provide information about the base object the user is currently working with.

   - **Disable In Show Mode** : Select `True` if the button should only be available to the users with ReadWrite access permissions to the object. Select `False` if the button should be available to any user accessing the object's object profile.

   - **Icon**: Select an icon to display on the button. The icon should not be larger than 22 x 22 pixel.

   - **Hint**: Enter text for a tooltip. The tooltip will be displayed when the user points to the button.

4) To sort the order of the buttons created for the object profile, click the object view ![icon] that the buttons have been created for to activate the attribute window. In the **Buttons** attribute, click the **Browse** ![icon] button. An editor opens that allows the sequence of the buttons in the toolbar to be defined via the **Up/Down** ![icon] button. Sort the buttons and click **OK** to close the editor.

5) In the toolbar, click the **Save** ![save icon] button.

Executing and Controlling ADIF via the ADIF Jobs Administration and My ADIF Jobs functionalities in the Alfabet User Interface

You can start ADIF import and export and control the success of execution directly in the Alfabet user interface.

Two business functions are available for execution and control of ADIF jobs:

- The business function **ADIF Jobs Administration** (`ADMIN_AdifJobs`) for execution and test of ADIF import or export and for checking the success of ADIF imports and ADIF exports.

- The business function **My ADIF Jobs** (`USER_AdifJobs`) for checking the success of ADIF import or export executed by the current user.

The following information is available:

- [Configuring Availability of ADIF Execution and Control in the User Interface](#)

    - [Configuring Asynchronous Execution](#)

        - [Configuring Access Permissions to Folders in the Internal Document Selector for Asynchronous Export to File](#)

        - [Configuring a User to Execute Self-Reflective Events](#)

- [Controlling Success of Executed ADIF Jobs](#)

- [Executing ADIF](#)

- [Testing ADIF Scheme Execution](#)

Configuring Availability of ADIF Execution and Control in the User Interface

The following configuration is required to administrate and/or execute ADIF jobs in the Alfabet user interface:

- Make the functionalities available to the user via the user profile, guide page or guide view configuration.

    > ⓘ For information about adding business functions directly to user profiles, see *Making Functionalities Accessible to a User Profile*. For information about adding business functions to user profiles via guide pages or guide views, see the reference manual *Designing Guide Pages for Alfabet*.

- The availability of ADIF jobs in the functionalities is controlled per ADIF scheme. By default an ADIF scheme and the ADIF jobs executed for the ADIF scheme are not visible on the Alfabet user interface. Set the attribute **Alfabet User Interface Behavior** to one of the following values to enable execution and administration or administration only via the Alfabet user interface:

    - `VisibleNotExecutable`: Information about the execution of ADIF jobs based on the ADIF scheme will be visible in the **ADIF Jobs Administration** and **My ADIF Jobs** functionalities. If

the user selects the ADIF scheme in the **ADIF Jobs Administration** functionality, the buttons for execution of the ADIF import or ADIF export will be deactivated.

- `VisibleExecutable`: Information about the execution of ADIF jobs based on the ADIF scheme will be visible in the **ADIF Jobs Administration** and **My ADIF Jobs** functionalities. The user can execute an ADIF import or ADIF export via button interaction in the **ADIF Jobs Administration** functionality.

> ⚠ It is not possible to define parameters for ADIF jobs executed via the user interface. ADIF jobs that depend on parameters handed over during execution should not be set to `VisibleExecutable`.

- ADIF jobs can be configured synchronously or asynchronously via the ADIF Jobs Administration functionality. Asynchronous execution is recommended to avoid problems caused by long running ADIF jobs blocking the current user session. Asynchronous execution requires the implementation of a set of preconditions described below in the section.

Configuring Asynchronous Execution

ADIF jobs can be executed asynchronously via the **ADIF Jobs Administration** or **My ADIF Jobs** functionalities. Synchronous execution blocks the current user session as long as the ADIF job is running. It is therefore recommended to use asynchronous execution.

In addition to the configuration described in this section, the following preconditions apply to enable asynchronous execution:

- A running Alfabet Server must be connected to the same database as the Alfabet Web Application providing the asynchronous ADIF execution functionality on the user interface.

- The RESTful services of the Alfabet Web Application must be activated and configured as described in the chapter *Activating the Alfabet RESTful API on Server Side* of the reference manual *Alfabet RESTful API* with **Has ADIFAPIInvocation Access** permissions granted in the server alias of the Alfabet Web Application.

- [Configuring Access Permissions to Folders in the Internal Document Selector for Asynchronous Export to File](#)

- [Configuring a User to Execute Self-Reflective Events](#)

Configuring Access Permissions to Folders in the Internal Document Selector for Asynchronous Export to File

ADIF export to file via asynchronous execution can exclusively target the **Internal Document Selector**. Export of data to the local file system is not available for asynchronous execution.

The asynchronous export requires setting of explicit access permissions for folders of the **Internal Document Selector**.

To define the IDOC folders that are accessible via asynchronous execution:

1) In the **Presentation** tab of Alfabet Expand, expand the explorer node **XML Objects** > **Administration**.

2) Right-click on the XML object **IDocManagerConfiguration** in the explorer and select **Edit XML** from the context menu. The XML object opens in the middle pane.

3) Enter the following code:

```
<IDocManager>
    <UseCase Name="JobSchedule" >
        <Folder Name="FolderShowName" Path="IDOC:\Path" />
    </UseCase>
</IDocManager>
```

The XML element `UseCase` can have multiple child elements `Folder`, each defining one folder that shall be accessible via the job scheduler. Define the folder with the following XML attributes of the XML element `Folder`:

- `Name`: Define a caption that shall be displayed for the folder in the editor for selecting the target file for ADIF export to file. The name does not have to be identical to the name of the folder in the **Internal Document Selector**.

- `Path`: Define the path to the folder in the **Internal Document Selector**. The path must start with `IDOC:\` and end with the name of the selected folder. Backslashes must be used between folder names. The ADIF export mechanism can read and write in the defined folder and all subordinate folders of the defined folder. All subordinate folders are also listed for selection in the editor for selecting the target file for ADIF export to file.

4) In the toolbar, click the **Save** button.

Configuring a User to Execute Self-Reflective Events

The user selected for execution of events of the type `SelfReflective` will be used for authentication of RESTful service calls to the RESTful API of the Alfabet Web Application for both wakeup events involved in job scheduling and for all other event based activities that involve execution via the RESTful API.

> For more information about triggering actions via event templates, see the chapter *Configuring Events*.

It is recommended to create a user that is exclusively used for executing events via the RESTful services. The user can be excluded from access to any objects except for executing job schedules, asynchronously starting ADIF jobs via the user interface, and start of workflows, RESTful service calls and/or execution of ADIF jobs via events based on event templates, but it is also possible to use an existing user.

The user must be a named user with at least one user profile assigned. The user profile is not used for evaluation of access permissions. A `ReadOnly` user profile is sufficient to execute the RESTful services in the context of events, asynchronous ADIF execution and job scheduling.

Only one user can be selected for execution of events of the type `SelfReflective`. If you assign this functionality to a user while another user has already been selected to execute self-reflective events, the setting is removed from that user when it is set for the user you are currently assigning it to. Therefore, if a user has already been defined in the context of the activation of any other functionality, you should add

the required access permissions described below to this user instead of creating another user to make sure that the permissions for the functionality the user was created for is maintained.

To create a user for execution of self-reflective events in the **User Administration** functionality on the Alfabet user interface:

> ℹ️ The same functionality is also available via the Alfabet Administrator. For information about accessing the user management functionalities of the Alfabet Administrator see *Functionalities Available via the Expanded Explorer of the Connected Alias* in the reference manual *System Administration*.

1) In the toolbar of the **User Administration** view, select **New** > **Create New User**. An editor opens.

2) In the editor, define the following:

   Basic Data tab:

   - **Name**: Enter a meaningful name for the user. The user is a technical user. You can either assign the name of an existing person to it or give it a name that indicates that this is a virtual person defined to execute a functionality.

   - **User Name**: Enter a user name. The user name is used by the RESTful services to identify the user.

   - **Type** : Select `NamedUser`.

   API Permissions tab:

   - **Has API V2 Access**: Select the checkbox.

   - **API Access Options**: Make sure that the following permission are selected:

     - **Has ADIFAPIInvocation Access** for the asynchronous execution of ADIF schemes via the user interface.

   - **Generate API Password** : Click the button. The API Password field is filled. For events of the type `Query`, copy the **API User Name** and the **API Password**. These have to be entered into the specification in the XML object `AlfabetIntegrationConfig` described below in the section *Configuring the Connection Parameters in the XML Object AlfabetIntegrationConfig*.

3) Click **OK** to close the editor. The new user is added to the table in the **User Administration** view.

4) Select the user in the table and select **Action** > **Set as Executes Self-Reflective Events User** in the toolbar.

5) Select the user in the table and click the **Navigate** 〉 button.

6) In the object profile of the user, click **Assigned User Profiles**.

7) In the toolbar of the **Assigned User Profiles** view, click **New** > **Assign User Profiles**.

8) In the selector, select a user profile and click **OK** to assign it to the user. For a user that is exclusively used for events, it is recommended to use a ReadOnly user profile.

The configuration described above is the configuration required for the asynchronous execution of ADIF jobs only. In addition, you can set any other properties of the user. For security reasons you might also consider to assign a login password to the user although the user password is not required for the **Job Schedule** functionality. For information about the available configuration options for users, see *Defining and Managing Users* in the reference manual *User and Solution Administration*.

Controlling Success of Executed ADIF Jobs

In the **ADIF Job Administration** and **My ADIF Jobs** functionalities, executed ADIF jobs are listed in an expandable dataset that shows information about ADIF job execution in the structure that is defined for the ADIF schemes in Alfabet Expand:

- The first level displays the root node and informs about the overall number of executed events matching the search conditions defined in the filter of the view.

- The second level displays ADIF scheme groups and ADIF schemes that are located directly under the root node.

- The third level displays the ADIF schemes in the ADIF group, or, for ADIF schemes located directly under the root node, the list of executed jobs.

- The fourth level displays the execution of ADIF jobs for ADIF schemes assigned to a group. The level has two sections **Executed Job/s** and **Started Job/s**. If an ADIF job is started asynchronously, the ADIF job is listed in the **Started Job/s** section during execution. After execution is finished, it is moved to the **Executed Job/s** section. Please note that asynchronous ADIF jobs are not listed in any of the sections as long as they are scheduled for execution at the server but execution has not been started yet. Synchronous execution of ADIF schemes inhibits reload of the view and therefore the jobs will never be visible in the **Started Job/s** section.

If ADIF jobs have been executed for an ADIF scheme one or multiple times, each execution is listed as a separate row under the heading **Executed Job/s** with the following information about the ADIF job execution:

- **ADIF Scheme Group**: The name of the ADIF scheme group the ADIF scheme is assigned to.

- **ADIF Scheme**: The caption of the ADIF scheme the ADIF job is executing.

- **ADIF Scheme Type**: Displays **EXPORT** for execution of an ADIF export scheme or **IMPORT** for execution of an ADIF import scheme.

- **Current Status**: Displays the overall status of the ADIF job execution:

  - **Started**: The ADIF job is currently executed. The ADIF job is listed in the section **Started Job/s**.

  - **Success**: The ADIF job has been successfully executed.

  - **Warning**: The ADIF job was executed, but a warning was written to the log file because a non-critical function failed.

  - **Failed**: The ADIF job has returned an error during execution.

- **Job Session**: The unique session ID of the ADIF job.

- **Start Time**: The time when the ADIF job execution started.

- **End Time**: The time when the ADIF job execution finished.

- **Executed By**: The way the ADIF job execution was started. The column returns one of the following:

  - `Expand ADIF Debugger`: The ADIF job was executed via the ADIF debugger of Alfabet Expand for testing purposes.

  - `User Interface`: The ADIF job execution was started from the Alfabet user interface via a button interaction. This is also displayed for test job execution on this view via the **Run Job** > **Non-Persistent Test Job** options.

  - `Rest API`: The ADIF job was triggered via a RESTful service call to the Alfabet RESTful API.

  - `Meta-Model Update`: The ADIF job was executed automatically during a meta-model update.

  - `Database Restore`: The ADIF job was executed automatically during a database restore from an Alfabet ADBZ file.

- **User Name**: The user name of the user that started the ADIF job. Please note the following about the user starting the ADIF job:

  - For ADIF jobs executed via a meta-model update or database restore that was started via the Alfabet Administrator the user information is empty because a user login is not required for the Alfabet Administrator.

  - For ADIF jobs executed via a RESTful service call to the endpoints `adifimport` or `adifexport` of the Alfabet RESTful API, the authentication user for the RESTful service call is identical with the user executing the ADIF job. This applies to RESTful service calls from an external RESTful client as well as for RESTful service calls triggered via events.

A filter is available to reduce the dataset to relevant content. Set the following filters fields on top of the table and click **Update** to view only data matching your filter settings:

- **ADIF Scheme Type**: Select **IMPORT** to view ADIF import schemes and ADIF jobs for ADIF export schemes only or **EXPORT** to view ADIF export schemes and ADIF jobs for ADIF export schemes only.

- **ADIF Schemes**: Select one or multiple ADIF schemes from the multi-select drop-down list to limit the display to the selected ADIF schemes and the ADIF jobs executed for the schemes.

- **ADIF Job State**: Select one of the following:

  - **Started** to view only currently running ADIF jobs.

  - **Success** to view only successfully executed ADIF jobs.

  - **Warning** to view only ADIF jobs executed with a warning message.

  - **Failed** to view only ADIF jobs terminated with an error.

  - **Execution Forcefully Terminated** to view only ADIF jobs that failed because the executing Alfabet Server was shut down either planned or forcefully during execution or the server thread for execution was forcefully terminated.

- **Start Date After**: Select a date from the calendar to view only ADIF jobs started at or after the selected date.

- **Start Date Before**: Select a date from the calendar to view only ADIF jobs started at or before the selected date.

To view the log information about an executed ADIF job:

1) In the toolbar, click the **Show Log** button.

2) In the window that opens, click the **Download** button. The file is downloaded via the download mechanisms of your browser.

(i) The button **Job Details** is only applicable for ServiceNow® integration. It provides information about the IDs of the log information returned by ServiceNow® that during execution of the ADIF job in verbose mode. The log information is stored in a table `ALFA_ADIF_SESSION_DETAIL` in the Alfabet database that is not visible on the Alfabet user interface or in Alfabet Expand. To see the log information, you can built a simple configured report based on the following query and export the information.

```
SELECT REFSTR,SESSION_ID,SCHEME_NAME,DETAIL_ID,IN_CONTENT,OUT_CONTENT
FROM ALFA_ADIF_SESSION_DETAIL
```

You can delete information about ADIF job execution if it is not required any longer to keep the information stored in the underlying database table small and to enhance the clarity of information in the **ADIF Jobs Administration** and **My ADIF Jobs** functionalities.

To delete ADIF job session information, select one or multiple ADIF job information in the table and click the **Delete** 🗑 button. The selected ADIF job session information is irrevocably deleted from the Alfabet database.

Executing ADIF

You can start ADIF jobs configured as executable via the user interface in the **ADIF Jobs Administration** functionality and not configured to require setting of mandatory parameters during execution.

ADIF jobs are executed asynchronously. They are queued for execution via an Alfabet Server. The execution might be delayed by other queued ADIF jobs. You can work on the Alfabet user interface in the current session while the ADIF job is queued and executed. The job will not be listed in the table of the **ADIF Jobs Administration** functionality while it is queued. It is listed in the section **Started Job/s** during execution and in the section **Executed Job/s** when finished. The user triggering the execution will also be informed about the success of the execution via an event feedback message.

To start execution of an ADIF job:

1) In the table, select the ADIF scheme that you would like to execute.

2) If the toolbar, click one of the following:

- **Run Job Asynchronously via Server** > **Execute Import Job Asynchronously**: The ADIF import job will be executed asynchronously and only errors and warnings and the execution start and end time will be written to the log file.

- **Run Job Asynchronously via Server** > **Execute Import Job Asynchronously with Verbose Logging**: The ADIF import job will be executed asynchronously and detailed information will be written to the log file during execution.

- **Run Job Asynchronously via Server** > **Execute Export Job Asynchronously**: The ADIF export job will be executed asynchronously and only errors and warnings and the execution start and end time will be written to the log file.

- **Run Job Asynchronously via Server** > **Execute Export Job Asynchronously with Verbose Logging**: The ADIF export job will be executed asynchronously and detailed information will be written to the log file during execution.

> (i) If the button options are deactivated, the selected ADIF scheme is configured to be visible but not executable on the Alfabet user interface.

3) If you are executing an ADIF import scheme that imports data from file, a file selector window opens. Select the file that includes the data for upload and click **Upload**.

4) If you are executing an ADIF export scheme that export data to a file, the export file is uploaded to the **Internal Document Selector** and will be available via the **Internal Documents** functionality. A dialog for selection of the allowed file locations in the Internal Document Selector opens. Select the checkbox of the target folder in the list of folder in the **IDOC Folder to Export** table and optionally specify a file name in the **Export File Name** field. The file name must have the extension.zip. If you do not specify a file name, the execution will generate a ZIP file with the name <NameOfADIFScheme>_<unformatted_timestamp>.zip.

Testing ADIF Scheme Execution

You can start all ADIF jobs configured as executable via the user interface in the **ADIF Jobs Administration** functionality in test mode. In test mode, information written to database tables is not persistent, but changes are reverted as last action of the ADIF job execution. The consequences of import can be tested via the log file without any risk to your data. For ADIF jobs exporting data to files, the test mode is not applicable. Using the test options will create the exported file as in normal execution mode.

1) In the table, select the ADIF scheme that you would like to execute.

2) If the toolbar, click any of the following:

- **Run Job Asynchronously via Server** > **Asynchronously Non-Persistent Import Test Job**: The ADIF import job will be executed asynchronously without persistently changing the data in the database and only errors and warnings and the execution start and end time will be written to the log file.

- **Run Job Asynchronously via Server** > **Asynchronously Non-Persistent Import Test Job with Verbose Logging**: The ADIF import job will be executed asynchronously without persistently changing the data in the database and detailed information will be written to the log file during execution.

- **Run Job Asynchronously via Server** > **Asynchronously Non-Persistent Export Test Job**: The ADIF import job will be executed asynchronously without persistently changing the data in the database and only errors and warnings and the execution start and end time will be written to the log file.

- **Run Job Asynchronously via Server** > **Asynchronously Non-Persistent Export Test Job with Verbose Logging**: The ADIF import job will be executed asynchronously without persistently changing the data in the database and detailed information will be written to the log file during execution.

3) If you are testing execution of an ADIF import scheme that imports data from file, a file selector window opens. Select the file that includes the data for upload and click **Upload**.

4) If you are executing an ADIF export scheme that export data to a file, the test mode will make no difference to the normal execution mode.

- The results are written into a persistent file. The export file is uploaded to the **Internal Document Selector** and will be available via the **Internal Documents** functionality. A dialog for selection of the allowed file locations in the **Internal Document Selector** opens. Select the checkbox of the target folder in the list of folder in the **IDOC Folder to Export** table and optionally specify a file name in the **Export File Name** field. The file name must have the extension.zip. If you do not specify a file name, the execution will generate a ZIP file with the name <NameOfADIFScheme>_<unformatted_timestamp>.zip.

Scheduling ADIF Jobs for Execution at a Defined Time With the Job Schedule Functionality

The functionality **Job Schedule** (`JobSchedule`) can be made available on the Alfabet user interface for administrative users. With this functionality, users can schedule ADIF jobs for either one time execution at a defined date and time or recurrent execution at defined times and intervals for a defined period of time.

In addition, the **Job Schedule** functionality can be used to batch delete old delete ADIF session information from the `ADIF_SESSION` table in the Alfabet database to reduce the size of the database and to enhance the clarity of information in the **ADIF Jobs Administration** and **My ADIF Jobs** functionalities.

The job scheduler requires a running Alfabet Server to be connected to the same database than the Alfabet Web Application providing the job scheduler on the user interface.

When a user creates a job schedule, the information about the scheduling is stored as an object of the object class Alfabet job schedule and a wakeup event is stored for the next time the job shall be executed.

The Alfabet Server scans the scheduled events in regular intervals for scheduled jobs that are due to be executed. If a job is due, the Alfabet Server puts the job into the queue of jobs to be executed.

The functionality must first be activated via configuration in Alfabet Expand.

The following information is available:

- Preconditions for Using the Job Scheduler

  - Creating Categories for the Job Schedule Use Case

  - Configuring the ADIF Scheme to be Executable via the Job Schedule Functionality

  - Configuring Access Permissions to Folders in the Internal Document Selector for the Job Schedule

  - Configuring a User to Execute Self-Reflective Events

  - Changing the Interval for Checking the Queues of Scheduled Events and ADIF Jobs

- Scheduling ADIF Jobs via the Job Scheduler Functionality

  - Creating a Job Schedule for ADIF Export

  - Creating a Job Schedule for ADIF Import

- Creating a Job Schedule for Batch Deletion of Old ADIF Session Information

Preconditions for Using the Job Scheduler

In addition to the configuration described in this section, the following preconditions apply:

- The job scheduler requires a running Alfabet Server to be connected to the same database as the Alfabet Web Application providing the **Job Schedule** functionality on the user interface.

- The RESTful services of the Alfabet Web Application must be activated and configured as described in the chapter *Activating the Alfabet RESTful API on Server Side* of the reference manual *Alfabet RESTful API*. The server alias of the Alfabet Web Application must grant **Has ADIFAPIInvocation Access** to start ADIF jobs via the **Job Schedule** functionality.

The following configurations are relevant for activating and configuring the job scheduler:

- Creating Categories for the Job Schedule Use Case

- Configuring the ADIF Scheme to be Executable via the Job Schedule Functionality

- Configuring Access Permissions to Folders in the Internal Document Selector for the Job Schedule

- Configuring a User to Execute Self-Reflective Events

- Changing the Interval for Checking the Queues of Scheduled Events and ADIF Jobs

Creating Categories for the Job Schedule Use Case

ADIF schemes can only be scheduled via the **Job Schedule** functionality if they are assigned to a category defined for the functionality in the XML object **UseCaseCategories**.

To set the categories for the **Job Schedule** functionality in the XML object **UseCaseCategories**:

1) In the **Presentation** tab of Alfabet Expand, expand the explorer node **XML Objects**.

2) Right-click on the XML object **UseCaseCategories** in the explorer and select **Edit XML** from the context menu. The XML object opens in the middle pane.

3) Enter the following code as child element of the XML element **UseCaseCategories**:

```
<UseCaseInfo UseCase="JobScheduler">
    <ScopeInfo Scope="ADIF" Categories="CommaSeparatedListOfCategories" />
</UseCaseInfo>
```

The XML attribute Categories must be set to either one category name or a comma separated list of category names. Please note that a category name is a technical name and should not contain special characters or whitespaces. All category names that are defined can be used in the **Category** attribute of an ADIF scheme to activate scheduling for the ADIF scheme.

Private ADIF schemes are delivered with Alfabet to trigger for example automated translation or import or export in the scope of an integration interface. If you would like to schedule execution of these private ADIF schemes, the list of categories must include the respective category that is preset for the ADIF scheme. The following category names are set for private ADIF schemes:

- `Translation` for all ADIF jobs for batch processing associated with automated translation.

- `Technopedia` for the ADIF import scheme `ALFABET_TECHNOPEDIA_UPDATE`.

- CentraSite for all ADIF jobs that are associated with CentraSite.

- APIGateway for the ADIF import scheme Alfabet_APIGateway_Synchronization.

- Apigee for the ADIF import scheme Alfabet_Apigee_Synchronization.

- APIPortal for the ADIF import scheme Alfabet_APIPortal_Synchronization.

4) In the toolbar, click the **Save** 💾 button.

Configuring the ADIF Scheme to be Executable via the Job Schedule Functionality

The ADIF Job must meet the following preconditions to be scheduled via the **Job Schedule** functionality:

- It must be assigned to one of the categories defined for the use case JobScheduler and the scope ADIF in the XML object **UseCaseCategories**.

  > ℹ️ For information about defining category names in the XML object **UseCaseCategories**, see *Creating Categories for the Job Schedule Use Case*.

- The ADIF scheme must not include parameters defined in compatibility mode or with the data type StringArray or ReferenceArray.

To activate job scheduling for an ADIF import scheme:

1) In the **ADIF** tab of Alfabet Expand, click the ADIF scheme.

2) In the attribute window, select one of the categories defined for the use case JobScheduler and the scope ADIF in the XML object **UseCaseCategories**.

Configuring Access Permissions to Folders in the Internal Document Selector for the Job Schedule

ADIF import scheduled via the job scheduler can be performed from a file located in the **Internal Document Selector** of the Alfabet database. This method is optional. Import can also be performed from a file on the local file system. ADIF export to file via the job scheduler can exclusively target the **Internal Document Selector**. Export of data to the local file system is not available for scheduled jobs.

The job scheduler can only access folders of the **Internal Document Selector** with explicit access permissions for the job scheduler.

To define the IDOC folders that are accessible via scheduled ADIF jobs:

1) In the **Presentation** tab of Alfabet Expand, expand the explorer node **XML Objects** > **Administration**.

2) Right-click on the XML object **IDocManagerConfiguration** in the explorer and select **Edit XML** from the context menu. The XML object opens in the middle pane.

3) Enter the following code:

```
<IDocManager>
    <UseCase Name="JobSchedule" >
        <Folder Name="FolderShowName" Path="IDOC:\Path" />
```

```
        </UseCase>

    </IDocManager>
```

The XML element `UseCase` can have multiple child elements `Folder,` each defining one folder that shall be accessible via the job scheduler. Define the folder with the following XML attributes of the XML element `Folder`:

- `Name`: Define a caption that shall be displayed for the folder in the editor for scheduling ADIF jobs. The name does not have to be identical to the name of the folder in the **Internal Document Selector**.

- `Path`: Define the path to the folder in the **Internal Document Selector**. The path must start with `IDOC:\` and end with the name of the selected folder. Backslashes must be used between folder names. The job scheduler can read and write in the defined folder and all subordinate folders of the defined folder.

4) In the toolbar, click the **Save** button.

Configuring a User to Execute Self-Reflective Events

The user selected for execution of events of the type `SelfReflective` will be used for authentication of RESTful service calls to the RESTful API of the Alfabet Web Application for both wakeup events involved in job scheduling and for all other event based activities that involve execution via the RESTful API.

(i) For more information about triggering actions via event templates, see the chapter *Configuring Events*.

It is recommended to create a user that is exclusively used for executing events via the RESTful services. The user can be excluded from access to any objects except for executing job schedules, asynchronously starting ADIF jobs via the user interface, and start of workflows, RESTful service calls and/or execution of ADIF jobs via events based on event templates, but it is also possible to use an existing user.

The user must be a named user with at least one user profile assigned. The user profile is not used for evaluation of access permissions. A ReadOnly user profile is sufficient to execute the RESTful services in the context of events.

Only one user can be selected for execution of events of the type `SelfReflective`. If you assign this functionality to a user while another user has already been selected to execute self-reflective events, the setting is removed from that user when it is set for the user you are currently assigning it to. Therefore, if a user has already been defined in the context of the activation of any other functionality, you should add the required access permissions described below to this user instead of creating another user to make sure that the permissions for the functionality the user was created for is maintained.

To create a user for execution of self-reflective events in the **User Administration** functionality on the Alfabet user interface:

(i) The same functionality is also available via the Alfabet Administrator. For information about accessing the user management functionalities of the Alfabet Administrator see *Functionalities Available via the Expanded Explorer of the Connected Alias* in the reference manual *System Administration*.

1) In the toolbar of the **User Administration** view, select **New** > **Create New User**. An editor opens.

2) In the editor, define the following:

Basic Data tab:

- **Name**: Enter a meaningful name for the user. The user is a technical user. You can either assign the name of an existing person to it or give it a name that indicates that this is a virtual person defined to execute a functionality.

- **User Name**: Enter a user name. The user name is used by the RESTful services to identify the user.

- **Type** : Select `NamedUser`.

API Permissions tab:

- **Has API V2 Access**: Select the checkbox.

- **API Access Options**: Make sure that the following permission are selected:

  - **Has ADIFAPIInvocation Access** for the execution of ADIF schemes via the **ADIF Import Job Schedule** and the **ADIF Export Job Schedule**.

  - **Has Batch Utilities API Access** for the execution of all other job schedules.

- **Generate API Password** : Click the button. The API Password field is filled. For events of the type `Query`, copy the **API User Name** and the **API Password**. These have to be entered into the specification in the XML object `AlfabetIntegrationConfig` described below in the section *Configuring the Connection Parameters in the XML Object AlfabetIntegrationConfig*.

3) Click **OK** to close the editor. The new user is added to the table in the **User Administration** view.

4) Select the user in the table and select **Action** > **Set as Executes Self-Reflective Events User** in the toolbar.

5) Select the user in the table and click the **Navigate** ⊳ button.

6) In the object profile of the user, click **Assigned User Profiles**.

7) In the toolbar of the **Assigned User Profiles** view, click **New** > **Assign User Profiles**.

8) In the selector, select a user profile and click **OK** to assign it to the user. For a user that is exclusively used for events, it is recommended to use a ReadOnly user profile.

The configuration described above is the configuration required for the **Job Schedule** functionality only. In addition, you can set any other properties of the user. For security reasons you might also consider to assign a login password to the user although the user password is not required for the **Job Schedule** functionality. For information about the available configuration options for users, see *Defining and Managing Users* in the reference manual *User and Solution Administration*.

Changing the Interval for Checking the Queues of Scheduled Events and ADIF Jobs

This configuration is optional. The default interval of the Alfabet Server for checking the wakeup events created for execution of job schedules for ADIF jobs due for execution is 500 milliseconds and the default interval of the Alfabet Server for checking the ADIF jobs queue and execute the next ADIF job is 3000 milliseconds. Optionally, you can adapt these values to your demands.

The configuration has to be performed in the tool Alfabet Administrator:

1) In the explorer of the Alfabet Administrator, click the **Alfabet Aliases** node.

2) In the table, select the server alias configuration that is used for starting the Alfabet Server.

3) In the toolbar, click the Edit [✎] button. The server alias editor opens.

4) Go to the **Server Settings** > **General** tab and edit the following attributes:

   - **ADIF Job Server Sleep Time (milliseconds)**: Enter the time between checking of the queue of ADIF jobs to be executed.

   - **Event Server Sleep Time (milliseconds)**: Enter the time between checking the event queue for scheduled wakeup events to be executed.

5) Click **OK** to save your changes.

Scheduling ADIF Jobs via the Job Scheduler Functionality

ADIF jobs are scheduled in the **Job Schedule** functionality.

The functionality lists all job schedules that have been defined. The table is an expandable dataset with 6 levels.

- The first level is the grouping level for the job schedule stereotype. There is one group for each job schedule stereotype. Currently, the following job schedule stereotypes are available:

   - **ADIF Export Job Schedule** for execution of ADIF export.

   - **ADIF Import Job Schedule** for execution of ADIF import.

   - **Rescan Indicator Job Schedule** to re-compute all automatically computed indicators.

   - **Rescan Color Rule Job Schedule** to update coloring based on color rules for new and changed objects.

   - **Full Text Search Utility Job Schedule** to update the full text search index to current changes in the Alfabet database.

   - **Alfa Batch Executor Job Schedule** for creating assignments and/or sending automatic email notifications to an object's authorized user if a monitor is triggered, an assignment approaches or reaches a defined due date, or an organizational or process change has occurred that affects an object.

   - **Alfa Workflow Job Schedule** for starting workflows configured to be started automatically, for closing workflow steps configured to be closed automatically, for automatic deletion of finished workflows, and for rescanning and updating roles and responsibilities for workflows and workflow steps.

   - **Publication Job Schedule** for publishing data based on a configured publication.

   - **Clear ADIF Session Content Job Schedule** for removing old entries from the ADIF_SESSION table.

- The second level displays all defined job schedules that belong to the job schedule stereotype.

- The third level is the grouping level for all events of the job schedule.

- The fourth level displays each execution of a job with the job schedule execution status.

- The fifth level is the grouping level for the job run status.

- The sixth level displays the job schedule execution status and provides the ability to open a log file returned from the executed ADIF scheme.

The table provides the following information about the execution of the job schedules:

- **Type**: The column displays the job schedule stereotype as group level for all stereotypes for that job schedules are available. The **Type** column displays **Events** as group level for all events of a job schedule. The **Type** column displays **Job Run Status** for the group level of the job target execution status.

- **Job Schedule Name**: The name of the job schedule. The name is defined by the user that creates the job schedule.

- **Stereotype**: The stereotype of the job schedule.

- **Next Run Time**: The time the job triggered by the job schedule will be executed because of this event.

- **Event Status**: The status of the event. The status can be one of the following:

    - **Pending**: The job is scheduled for execution at a later point of time.

    - **Executing**: The job is currently executed.

    - **Finished**: Job execution has been success. If an event has the status **Finished**, the scheduling of the job was finished successfully, and a **Job Run Status** section is added as subordinate information level. The execution status in the Job Run Status section should then be checked to see whether execution of the scheduled functionality has been successful, too.

    - **Error**: An error related to scheduling of the job or triggering of the functionality to be executed has occurred.

    - **Execution Error**: An error has occurred during execution of the functionality.

- **Occurrence Time**: The time when the job event was scheduled. A job event is scheduled directly after creation of the job schedule for the first execution. For recurrent schedules, new event is scheduled for the next execution as soon as the first execution is done.

- **Execution Time**: The time when the execution of the job triggered by the event was started.

- **Completion Time**: The time when the execution of the job triggered by the event has finished.

A filter is available on top of the table to search for specific events in the table:

- **Stereotype**: Select a job schedule stereotype to view only job schedules that belong to the selected stereotype.

- **Name**: Enter the name of a job schedule to view only events for the selected job schedule.

- **Job Schedule Event Status**: Select a job schedule status to view only events that are currently in the selected status. Please note that this filter is only applied to the event status of the scheduling events listed in the **Events** sections.

- **Execution After**: Enter a date to view only job schedule events executed at or after the selected date.

- **Execution Before**: Enter a date to view only job schedule events executed at or before the selected date.

You can do one of the following for existing job schedules.

- To view the log file of a job executed via an event, click the job run status in the table and click **Show Log** in the toolbar.

- To cancel a pending event, select the event in the table and click **New** > **Cancel Planned Job**.

- To cancel all pending events for a job schedule, select the job schedule in the table and click **New** > **Cancel All Planned Jobs**.

- After having cancelled pending events, you should select the job schedule in the table and click **New** > **Resume Job Schedule** to resume the execution.

- To schedule an event for immediate execution of the job independent from the defined execution time-table for the job schedule, select the job schedule in the table and click **New** > **Execute Job Schedule** in the toolbar. This option can be used to do one of the following:

  - If a job schedule has reached the end date of its execution cycle or has been configured to trigger a single execution, the ADIF job or batch job triggered by the job schedule can be scheduled for immediate execution using the **Execute Job Schedule** option.

  - If pending events for a job schedule were cancelled, the **Execute Job Schedule** option can be used instead of the **Resume Job Schedule** option to resume the execution. While the **Resume Job Schedule** option will resume the execution at the next scheduled execution date, the **Execute Job Schedule** option will execute the job immediately independent from the configured execution times in the job schedule and at the same time create an event for the execution at the next scheduled execution date.

- To delete a job schedule, select the job schedule in the table and click the **Delete** 🗑 button in the toolbar.

- To change a job schedule, select the job schedule in the table and click the **Edit** ✎ button in the toolbar.

  > ⓘ If you change a job schedule with a pending event, the pending event will be canceled and a new event will be scheduled that contains the changes you made in the editor.

- To create a new job schedule on basis of the data of an existing job schedule, select the job schedule that shall be used as template in the table and click **New** > **Create Job Schedule as Copy** in the toolbar.

To create a job schedule, do one of the following:

- [Creating a Job Schedule for ADIF Export](#)

- [Creating a Job Schedule for ADIF Import](#)

Creating a Job Schedule for ADIF Export

Please note that files resulting from scheduled ADIF export can only be exported to the **Internal Document Selector** of the Alfabet database. They can be downloaded from the **Internal Document Selector** in

the **Internal Documents** functionality or via a RESTful service call to the `idocdownload` end point of the Alfabet RESTful services.

To schedule an ADIF export job:

1) In the toolbar, click **New** > **Create Schedule**.

2) In the **Stereotype Selector** that opens, select **ADIF Export Job Schedule**.

3) In the editor, define the following attributes:

   Basic Data tab:

   - **Name**: Define a name for the job schedule. The job schedule will be listed in the table of the **Job Schedule** functionality with this name. The name must be unique. It is not possible to define two job schedules with the same name, even if the type of job schedule is different. This attribute is mandatory.

   - **Description**: Provide a meaningful description about the purpose of the job schedule.

   - **Verbose Logging**: Select the checkbox if additional information about the running process shall be logged. Verbose logging is in most cases not required and can lead to a decrease in performance. The default setting for this attribute is `False`.

   - **ADIF Export Scheme**: Select the ADIF export scheme that shall be executed via the job schedule from the drop-down list.

     > (i) Only ADIF schemes that are assigned to a job schedule category can be executed via the **Job Schedule** functionality. For more information see *Configuration Required for Scheduling ADIF Jobs* in the reference manual *Configuring Alfabet with Alfabet Expand*.

   - **Parameters**: If the selected ADIF export scheme includes parameter definitions, the attribute displays a table that informs about the name, type and default value for the defined parameters and whether the parameters are mandatory. Enter a parameter value at least for all parameters not having a default defined into the fields in the column **Parameter Value**. Note the following about the definition of parameter values:

     - Parameter values for data types like string or date are defined without single quotes at the beginning and end. If single quotes are required in the query for the data type, they will be automatically added by the ADIF mechanisms.

     - For the definition of dates enter the date in the **Value** field in the format defined in the culture setting of the language you are currently using to render the Alfabet user interface.

     - String array values must be separated with \r\n.

     - % can be used as wildcard in strings and texts. It is not allowed to define a wildcard in a value of a string array.

     - The definition of values for string arrays and reference arrays is currently not supported.

   Schedule tab:

   - **Schedule Time**: Enter the start time for the job execution into the **Start Time** field.

- **Recurrence Pattern**: Select one of the following check boxes and provide additional data, if applicable:

  - **Daily**: The interval between job executions is the number of days specified in the field **day(s)** as an integer.

  - **Weekly**: The interval between job executions is the number of weeks specified in the field **week(s) on** as an integer. In addition, select the checkbox of the day of the week that the job shall be executed for each recurrence.

  - **Monthly**: There are tow methods to select from. If you select the checkbox **Date-Based**, define a day of the month in the **day** field behind the checkbox. For example if you want the job to be executed on the fifteenth of each month, enter 15. If you select the checkbox **Weekday-Based**, you can schedule the job for a specific day of a specific week in the month. In the first field behind the checkbox, select the week of the month the job shall be executed. In the second field behind the checkbox, select the day of the week the job shall be executed. The interval between job executions is the number of weeks specified in the field **month(s)** field of the selected scheduling method.

  - **Once**: The job is only executed once.

- **Range of Recurrence**: This definition is mandatory. Select the start date for the job schedule from the calendar in the **Start Date** field. For the definition of the end of the recurrence period, either select **End after:** and define the maximum number of occurrences in the **Occurrences** field or select **End by:** and select the end date for the job scheduling from the calender in the field next to the checkbox entry. If you have selected **Once** in the **Recurrence Pattern** field, only define the date of execution via the **Start Date** field.

  File Info tab: This tab is only relevant for export to file.

- **File Name Base**: Enter the file name without extension for the ZIP file that shall contain the exported data. If you do not define the name, the file name will be the name of the executed ADIF export scheme in capital letters.

- **File Suffix**: Select from the drop-down list whether and how the **File Name Base** shall be complemented with a unique marker on export. This setting is relevant for regular exports that should each generate a file without overwriting the previously exported files. The file name can either be complemented with a timestamp or with a unique GUID.

- **IDOC Folder to Export**: Select the checkbox on the right of the folder in the **Internal Document Selector** the file shall be stored in. The list does not display the complete structure of the **Internal Document Selector** but only the folders defined to be used for job scheduling. The file names displayed in the **IDOC Folder to Export** field can deviate from file names in the **Internal Document Selector**, because a show name can be configured for the folders in Alfabet Expand.

  (i) For more information, see *Configuring Access Permissions to Folders in the Internal Document Selector for the Job Schedule* in the reference manual *Configuring Alfabet with Alfabet Expand*.

If no folder is selected, the file is saved in a folder ADIF_SYS on the root level of the **Internal Document Selector**.

  Execution Info tab:

This tab provides the ability to adapt the job scheduling to environmental conditions, like for example the scheduling of other jobs or maintenance windows during which execution shall be postponed.

- **Maximum Wait Time [min]**: Enter a maximum time in minutes that the job execution can be postponed if one of the following applies:

  - The execution of jobs for the current job schedule depends on the execution of one or multiple other jobs to be finished because the execution result of one job is required to be available to execute this job or because both jobs perform conflicting actions. If a job of the current schedule shall be executed while a job defined in the **Dependent Jobs** field is still running, the job will not be executed but re-scheduled for execution five minutes later until the dependent jobs are finished.

  - A job for the current job schedule cannot be executed because the current execution time lies within a maintenance window. Maintenance windows are time periods that are blocked to avoid interruption of the job by maintenance work. If a job is due within a maintenance window period, the job will not be executed immediately. It will be re-scheduled for execution one minute after the end of the maintenance window.

    > ⓘ Maintenance windows must have been configured for the Alfabet components in order to adapt job scheduling to the maintenance window via the **Maximum Wait Time [min]** setting. For information about the definition of maintenance windows, see *Defining Maintenance Windows* in the reference manual *Configuring Alfabet with Alfabet Expand* or *Defining Maintenance Windows for Scheduled Jobs* in the reference manual *System Administration*.

If the time between the originally due job execution and the end of the maintenance window or the end of execution of a dependent job exceeds the maximum wait time, job execution will be cancelled, and an error message is written into the log file. The next execution of the job is scheduled according to the settings in the **Schedule** tab of the job schedule editor.

- **Dependent Jobs**: If the execution of jobs for the current job schedule shall be postponed if another job is still running, click **New** > **Create New Job Dependency**. Select the job schedule the current job schedule depends on in the list and click **OK**. If a job is running for one of the dependent job schedules, the execution of the job for the current job schedule will be shifted to the scheduled time plus minutes until the execution of all dependent jobs is finished. If the maximum wait time defined with **Maximum Wait Time [min]** is exceeded while dependent jobs are still running, the execution of the current job is cancelled and the next execution scheduled according to the settings in the **Schedule** tab of the job schedule editor.

- **Expected Execution Time [min]**: If a maximum time for execution of the job is defined in minutes in this field, a warning is written into the log file available via the **Show Log** button if the execution of a job for the current schedule exceeds the maximum wait time. The job is nevertheless further executed.

- **Executing User**: Each time a scheduled job has been executed, the user that owns the job schedule will see a slide-in **Event Feedback** message window informing him/her whether the current execution was successful. If a result is available for download, a download link is displayed. By default, the slide-in message will be displayed to the user that created the job schedule when he/she is logged in with the same user profile as on creation of the job schedule. This default behaviour can be changed. Select a user from the drop-down list to show the **Event Feedback** message to that user instead. Please note that the selected user must be logged in with the user profile defined in the **Executing User Profile** attribute to receive the **Event Feedback** messages for this job schedule.

- **Executing User Profile**: The user profile that the user defined with the **Executing User** attribute must be logged in with to receive **Event Feedback** messages for execution of this job schedule. By default, message will be displayed to the user that created the job schedule when he/she is logged in with the same user profile as on creation of the job schedule. This attribute is mandatory if **Executing User** is defined.

4) Click **OK** to save your changes.

After having created the job schedule, you can use it as a template for creating new job schedules. Select the job schedule in the table and click **New** > **Create Job Schedule As Copy** from the toolbar.

Creating a Job Schedule for ADIF Import

For ADIF import from file, the files can either be located on the local file system in a folder accessible for the Alfabet components or in the Internal Document Selector of Alfabet.

To schedule an ADIF import job:

1) In the toolbar, click **New** > **Create Schedule**.

2) In the **Stereotype Selector** that opens, select **ADIF Import Job Schedule**.

3) In the editor, define the following attributes:

Basic Data tab:

- **Name**: Define a name for the job schedule. The job schedule will be listed in the table of the **Job Schedule** functionality with this name. The name must be unique. It is not possible to define two job schedules with the same name, even if the type of job schedule is different. This attribute is mandatory.

- **Description**: Provide a meaningful description about the purpose of the job schedule.

- **Verbose Logging**: Select the checkbox if additional information about the running process shall be logged. Verbose logging is in most cases not required and can lead to a decrease in performance. The default setting for this attribute is `False`.

- **Import Scheme**: Select the ADIF import scheme that shall be executed via the job schedule from the drop-down list.

  > (i) Only ADIF schemes that are assigned to a job schedule category can be executed via the **Job Schedule** functionality. For more information see *Configuration Required for Scheduling ADIF Jobs* in the reference manual *Configuring Alfabet with Alfabet Expand*.

- **Parameters**: If the selected ADIF import scheme includes parameter definitions, the attribute displays a table that informs about the name, type and default value for the defined parameters and whether the parameters are mandatory. Enter a parameter value at least for all parameters not having a default defined into the fields in the column **Parameter Value**. Note the following about the definition of parameter values:

  - Parameter values for data types like string or date are defined without single quotes at the beginning and end. If single quotes are required in the query for the data type, they will be automatically added by the ADIF mechanisms.

- For the definition of dates enter the date into the Value field in the format defined in the culture setting of the language you are currently using to render the Alfabet user interface.

- String array values must be separated with \r\n.

- % can be used as wildcard in strings and texts. It is not allowed to define a wildcard in a value of a string array.

- The definition of values for reference arrays is currently not supported.

Schedule tab:

- **Schedule Time**: Enter the start time for the job execution into the **Start Time** field.

- **Recurrence Pattern**: Select one of the following check boxes and provide additional data, if applicable:

  - **Daily**: The interval between job executions is the number of days specified in the field **day(s)** as an integer.

  - **Weekly**: The interval between job executions is the number of weeks specified in the field **week(s) on** as an integer. In addition, select the checkbox of the day of the week that the job shall be executed for each recurrence.

  - **Monthly**: There are two methods to select from. If you select the checkbox **Date-Based**, define a day of the month in the **day** field behind the checkbox. For example, if you want the job to be executed on the fifteenth of each month, enter 15. If you select the checkbox **Weekday-Based**, you can schedule the job for a specific day of a specific week in the month. In the first field behind the checkbox, select the week of the month the job shall be executed. In the second field behind the checkbox, select the day of the week the job shall be executed. The interval between job executions is the number of weeks specified in the field **month(s)** field of the selected scheduling method.

  - **Once**: The job is only executed once.

- **Range of Recurrence**: This definition is mandatory. Select the start date for the job schedule from the calendar in the **Start Date** field. For the definition of the end of the recurrence period, either select **End after:** and define the maximum number of occurrences in the **Occurrences** field or select **End by:** and select the end date for the job scheduling from the calender in the field next to the checkbox entry. If you have selected **Once** in the **Recurrence Pattern** field, only define the date of execution via the **Start Date** field.

File Info tab: this tab is only relevant for import from file.

- **File Location**: Select where the import ZIP file is located.

  - **System File**: The file is available on the local file system.

  - **System Folder**: The file is available in a defined folder. The latest file in the folder will be used to perform the import.

  - **IDOC File**: The file is available on the local file system.

  - **IDOC Folder**: The file is available in a defined folder of the Internal Document Selector. The latest file in the folder will be used to perform the import.

- **System File/Folder Path**: If the **File Location** is **System File**, enter the absolute path and name of the file to be imported. If the **File Location** is **System Folder**, enter the absolute path to the folder containing the import file.

- **Internal Documents**: If the **File Location** is **IDOC File**, select the checkbox behind the file in the internal document selector that contains the data to be imported. The file type must be ZIP. If the **File Location** is **IDOC Folder**, select the checkbox behind the folder that will contain the import file.

Execution Info tab:

This tab provides the ability to adapt the job scheduling to environmental conditions, like for example the scheduling of other jobs or maintenance windows during which execution shall be postponed.

- **Maximum Wait Time [min]**: Enter a maximum time in minutes that the job execution can be postponed if one of the following applies:

  - The execution of jobs for the current job schedule depends on the execution of one or multiple other jobs to be finished because the execution result of one job is required to be available to execute this job or because both jobs perform conflicting actions. If a job of the current schedule shall be executed while a job defined in the **Dependent Jobs** field is still running, the job will not be executed but re-scheduled for execution five minutes later until the dependent jobs are finished.

  - A job for the current job schedule cannot be executed because the current execution time lies within a maintenance window. Maintenance windows are time periods that are blocked to avoid interruption of the job by maintenance work. If a job is due within a maintenance window period, the job will not be executed immediately. It will be re-scheduled for execution one minute after the end of the maintenance window.

    (i) Maintenance windows must have been configured for the Alfabet components in order to adapt job scheduling to the maintenance window via the **Maximum Wait Time [min]** setting. For information about the definition of maintenance windows, see *Defining Maintenance Windows* in the reference manual *Configuring Alfabet with Alfabet Expand* or *Defining Maintenance Windows for Scheduled Jobs* in the reference manual *System Administration*.

If the time between the originally due job execution and the end of the maintenance window or the end of execution of a dependent job exceeds the maximum wait time, job execution will be cancelled and an error message is written into the log file. The next execution of the job is scheduled according to the settings in the **Schedule** tab of the job schedule editor.

- **Dependent Jobs**: If the execution of jobs for the current job schedule shall be postponed if another job is still running, click **New** > **Create New Job Dependency**. Select the job schedule the current job schedule depends on in the list and click **OK**. If a job is running for one of the dependent job schedules, the execution of the job for the current job schedule will be shifted to the scheduled time plus minutes until the execution of all dependent jobs is finished. If the maximum wait time defined with **Maximum Wait Time [min]** is exceeded while dependent jobs are still running, the execution of the current job is cancelled and the next execution scheduled according to the settings in the **Schedule** tab of the job schedule editor.

- **Expected Execution Time [min]**: If a maximum time for execution of the job is defined in minutes in this field, a warning is written into the log file available via the **Show Log** button if the execution of a job for the current schedule exceeds the maximum wait time. The job is nevertheless further executed.

- **Executing User**: Each time a scheduled job has been executed, the user that owns the job schedule will see a slide-in **Event Feedback** message window informing him/her whether the current execution was successful. If a result is available for download, a download link is displayed. By default, the slide-in message will be displayed to the user that created the job schedule when he/she is logged in with the same user profile as on creation of the job schedule. This default behaviour can be changed. Select a user from the drop-down list to show the **Event Feedback** message to that user instead. Please note that the selected user must be logged in with the user profile defined in the **Executing User Profile** attribute to receive the **Event Feedback** messages for this job schedule.

- **Executing User Profile**: The user profile that the user defined with the **Executing User** attribute must be logged in with to receive **Event Feedback** messages for execution of this job schedule. By default, message will be displayed to the user that created the job schedule when he/she is logged in with the same user profile as on creation of the job schedule. This attribute is mandatory if **Executing User** is defined.

4) Click **OK** to save your changes.

After having created the job schedule, you can use it as a template for creating new job schedules. Select the job schedule in the table and click **New** > **Create Job Schedule As Copy** from the toolbar.

Creating a Job Schedule for Batch Deletion of Old ADIF Session Information

ADIF session information older than a defined number of days can be batch deleted for a defined ADIF scheme via a job schedule. The information about job execution can either be deleted completely or deletion can be limited to the log file content while the information that the job was executed and the resulting execution state is retained.

To schedule a job for deletion of ADIF session information:

1) In the toolbar, click **New** > **Create Schedule**.

2) In the **Stereotype Selector** that opens, select **Clear ADIF Session Content Job Schedule**.

3) In the editor, define the following attributes:

   Basic Data tab:

   - **Name**: Define a name for the job schedule. The job schedule will be listed in the table of the **Job Schedule** functionality with this name. The name must be unique. It is not possible to define two job schedules with the same name, even if the type of job schedule is different. This attribute is mandatory.

   - **Description**: Provide a meaningful description about the purpose of the job schedule.

   - **ADIF Scheme**: Select the ADIF scheme for which job execution information shall be deleted from the Alfabet database. If no ADIF scheme is selected, the job execution information will be deleted for all ADIF schemes.

   - **Number of Days**: The number of days for that job execution information shall be retained. The number of days must be a positive integer. If you enter 1, information about job execution for the current date and the day before the current date is retained. If you enter 0, all information including the information of job execution on the current date is removed.

- **Level**: Select **Log** to remove only the log file content stored in the database while the information about the job execution date and the execution state is retained. Select **All** to remove all information about the job execution.

Schedule tab:

- **Schedule Time**: Enter the start time for the job execution into the **Start Time** field.

- **Recurrence Pattern**: Select one of the following check boxes and provide additional data, if applicable:

  - **Daily**: The interval between job executions is the number of days specified in the field **day(s)** as an integer.

  - **Weekly**: The interval between job executions is the number of weeks specified in the field **week(s) on** as an integer. In addition, select the checkbox of the day of the week that the job shall be executed for each recurrence.

  - **Monthly**: There are tow methods to select from. If you select the checkbox **Date-Based**, define a day of the month in the **day** field behind the checkbox. For example if you want the job to be executed on the fifteenth of each month, enter 15. If you select the checkbox **Weekday-Based**, you can schedule the job for a specific day of a specific week in the month. In the first field behind the checkbox, select the week of the month the job shall be executed. In the second field behind the checkbox, select the day of the week the job shall be executed. The interval between job executions is the number of weeks specified in the field **month(s)** field of the selected scheduling method.

  - **Once**: The job is only executed once.

- **Range of Recurrence**: This definition is mandatory. Select the start date for the job schedule from the calendar in the **Start Date** field. For the definition of the end of the recurrence period, either select **End after:** and define the maximum number of occurrences in the **Occurrences** field or select **End by:** and select the end date for the job scheduling from the calendar in the field next to the checkbox entry. If you have selected **Once** in the **Recurrence Pattern** field, only define the date of execution via the **Start Date** field.

Execution Info tab:

This tab provides the ability to adapt the job scheduling to environmental conditions, like for example the scheduling of other jobs or maintenance windows during which execution shall be postponed.

- **Maximum Wait Time [min]**: Enter a maximum time in minutes that the job execution can be postponed if one of the following applies:

  - The execution of jobs for the current job schedule depends on the execution of one or multiple other jobs to be finished because the execution result of one job is required to be available to execute this job or because both jobs perform conflicting actions. If a job of the current schedule shall be executed while a job defined in the **Dependent Jobs** field is still running, the job will not be executed but re-scheduled for execution five minutes later until the dependent jobs are finished.

  - A job for the current job schedule cannot be executed because the current execution time lies within a maintenance window. Maintenance windows are time periods that are blocked to avoid interruption of the job by maintenance work. If a job is due within a

maintenance window period, the job will not be executed immediately. It will be re-scheduled for execution one minute after the end of the maintenance window.

> ⓘ Maintenance windows must have been configured for the Alfabet components in order to adapt job scheduling to the maintenance window via the **Maximum Wait Time [min]** setting. For information about the definition of maintenance windows, see *Defining Maintenance Windows* in the reference manual *Configuring Alfabet with Alfabet Expand* or *Defining Maintenance Windows for Scheduled Jobs* in the reference manual *System Administration*.

If the time between the originally due job execution and the end of the maintenance window or the end of execution of a dependent job exceeds the maximum wait time, job execution will be cancelled and an error message is written into the log file. The next execution of the job is scheduled according to the settings in the **Schedule** tab of the job schedule editor.

- **Dependent Jobs**: If the execution of jobs for the current job schedule shall be postponed if another job is still running, click **New** > **Create New Job Dependency**. Select the job schedule the current job schedule depends on in the list and click **OK**. If a job is running for one of the dependent job schedules, the execution of the job for the current job schedule will be shifted to the scheduled time plus minutes until the execution of all dependent jobs is finished. If the maximum wait time defined with **Maximum Wait Time [min]** is exceeded while dependent jobs are still running, the execution of the current job is cancelled and the next execution scheduled according to the settings in the **Schedule** tab of the job schedule editor.

- **Expected Execution Time [min]**: If a maximum time for execution of the job is defined in minutes in this field, a warning is written into the log file available via the **Show Log** button if the execution of a job for the current schedule exceeds the maximum wait time. The job is nevertheless further executed.

- **Executing User**: Each time a scheduled job has been executed, the user that owns the job schedule will see a slide-in **Event Feedback** message window informing him/her whether the current execution was successful. If a result is available for download, a download link is displayed. By default, the slide-in message will be displayed to the user that created the job schedule when he/she is logged in with the same user profile as on creation of the job schedule. This default behavior can be changed. Select a user from the drop-down list to show the **Event Feedback** message to that user instead. Please note that the selected user must be logged in with the user profile defined in the **Executing User Profile** attribute to receive the **Event Feedback** messages for this job schedule.

- **Executing User Profile**: The user profile that the user defined with the **Executing User** attribute must be logged in with to receive **Event Feedback** messages for execution of this job schedule. By default, message will be displayed to the user that created the job schedule when he/she is logged in with the same user profile as on creation of the job schedule. This attribute is mandatory if **Executing User** is defined.

4) Click **OK** to save your changes.

After having created the job schedule, you can use it as a template for creating new job schedules. Select the job schedule in the table and click **New** > **Create Job Schedule As Copy** from the toolbar.

Chapter 8:   Predefined ADIF Schemes

Usually, ADIF import and export processes strongly depend on customer environments and requirements. Only few import processes are of general interest independent of the customer environment.

Predefined ADIF schemes are provided by Software AG if the demand for a specific import is of general interest. These predefined ADIF schemes are either private and therefore cannot be changed at all or protected, in which case you can edit only a subset of the attributes of the preconfigured child elements of the ADIF scheme. It is possible to add new ADIF elements to protected ADIF schemes. The ADIF elements added by the customer are public and completely editable. Private and protected ADIF schemes have a lock symbol indicating their editability status. The lock is orange for protected ADIF schemes and red for private ADIF schemes. For a protected ADIF scheme, editability of the attributes is visible in the attribute windows of the child elements only. Deactivated attributes are displayed in grey.
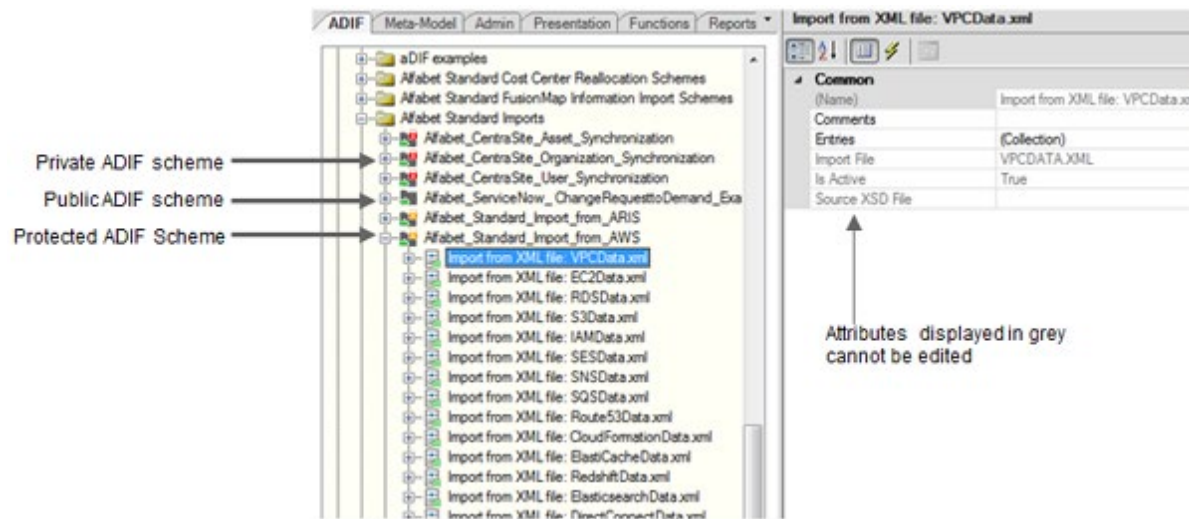
 FIGURE: Private, protected, and public ADIF schemes in the ADIF Explorer

Predefined ADIF schemes are delivered as part of the standard ADIF functionality and are documented as part of the triggered functionality or in the following. They can be adapted to the customer requirements prior to use. Execution is carried out with the ADIF console application in the same way as the execution of a customer-defined ADIF scheme.

The following ADIF import schemes are predefined:

Group	ADIF Import Scheme	Provided Functionality	For More Informaton See
Alfabet Standard Cost Centre Reallocation Schemes	`CC_Reallocate`	ADIF can be used to batch update budget costs and object assignment to a cost centre as well as cost allocation rates. For all imports via ADIF, cost re-allocation to objects assigned to the cost centres is not performed automatically after import of the changed cost allocation data. Cost re-allocation must either be performed manually or by running the pre-defined ADIF import scheme.	[Import Scheme to Automatically Reallocate Cost Centre Costs](#)
Alfabet Standard FusionMap Information Import Schemes	`FUSIONMAPINFO_IMPORT`		
Alfabet Standard Imports	`Alfabet_CentraSite_Asset_Synchronization`	If the interface for data synchronization with CentraSite is used, a standardized batch import of information about assets that are common to Alfabet and CentraSite can be performed via this predefined ADIF scheme. This ADIF scheme is read-only and cannot be customized.	For more information about data synchronization with CentraSite, see the section *Configuring Interoperability with CentraSite* in the reference manual *API Integration with Third-Party Components*.
Alfabet Standard Imports	`Alfabet_CentraSite_Organization_Synchronization`	If the interface for data synchronization with CentraSite is used, a standardized batch import of information about organizations that are common to Alfabet and CentraSite can be performed via this predefined ADIF scheme. This scheme is read-only and cannot be customized.	For more information about data synchronization with CentraSite, see the section *Configuring Interoperability with CentraSite* in the reference manual *API Integration with Third-Party Components*.

Group	ADIF Import Scheme	Provided Functionality	For More Informaton See
Alfabet Standard Imports	`Alfabet_CentraSite_User_Synchronization`	If the interface for data synchronization with CentraSite is used, a standardized batch import of information about users that are common to Alfabet and CentraSite can be performed via this predefined ADIF scheme. This scheme is read-only and cannot be customized.	For more information about data synchronization with CentraSite, see the section *Configuring Interoperability with CentraSite* in the reference manual *API Integration with Third-Party Components*.
Alfabet Standard Technopedia Interoperability Schemes	`ALFABET_TECHNOPEDIA_UPDATE`	Vendor products/components as well their vendors can be updated based on the data in Technopedia®, a categorized repository of information about enterprise hardware and software. An update can be performed by executing the standard ADIF import scheme. The import scheme is not customizable. The import scheme reads the configuration specified in the XML object **TechnopediaConfig**. ⓘ The reference of a vendor product to an ICT object will be updated during the execution of the standard ADIF schemes available for Alfabet Technopedia interoperability if both the vendor product as well as the ICT object originated from Technopedia and the Technopedia product version/release associated with the vendor product references a Technopedia product that is different from the Technopedia product referenced by the ICT object.	For more information about Technopedia interoperability, see the section *Configuring Interoperability with Technopedia* in the reference manual *API Integration with Third-Party Components*.
Alfabet Standard Imports	`Alfabet_APIGateway_Synchronization`	APIs in webMethods API Gateway may be imported to Alfabet by means of an ADIF import scheme. Technical services cannot be exported via an ADIF scheme to webMethods API Gateway. If the import of APIs via ADIF is specified in the XML object **APIGatewayConfig**, then the predefined ADIF scheme	For more information about the configuration required to import and map APIs from webMethods API Gateway to Alfabet, see the section *Configuring Interoperability with*

Group	ADIF Import Scheme	Provided Functionality	For More Informaton See
		`Alfabet_APIGateway_Synchronization` must be executed to trigger the import. The data from webMethods API Gateway will be imported to temporary tables in ADIF. Further configuration is required in order to update the Alfabet database tables with the data in the temporary database tables.	*webMethods API Gateway* in the reference manual *API Integration with Third-Party Components*.
Alfabet Standard Imports	`Alfabet_APIPortal_Synchronization`	APIs in webMethods API Portal may be imported to Alfabet by means of an ADIF import scheme. Technical services cannot be exported via an ADIF scheme to webMethods API Gateway. If the import of APIs via ADIF is specified in the XML object **APIPortalConfig**, then the predefined ADIF scheme `Alfabet_APIPortal_Synchronization` must be executed to trigger the import. The data from webMethods API Portal will be imported to temporary tables in ADIF. Further configuration is required in order to update the Alfabet database tables with the data in the temporary database tables.	For more information about the configuration required to import and map APIs from webMethods API Portal to Alfabet, see the section *Configuring Interoperability with webMethods API Portal* in the reference manual *API Integration with Third-Party Components*.
Alfabet Instance Automated Translations	`Get_Instance_Automated_Translations_For_Empty_Texts`	This private ADIF scheme retrieves automated translations for permissible object class properties that have a value in the primary language and are empty in one of the secondary languages. This ADIF scheme can be executed for object classes for which the **Enable Data Translation** attribute is set to `True`. and that have at least one property set to `True` in the **Property Automated Translation** attribute.	For more information about the automated data translation capability as well as the configuration required to execute automated translations, see the section *Configuring the Translation of Object Data* in the chapter *Localization and Multi-Language Support for the Alfabet Interface* in the reference manual *Configuring Alfabet with Alfabet Expand*.
Alfabet Instance	`GetAutomatedTranslations_For_Instance`	This private ADIF scheme retrieves automated translations for permissible object class properties reusing existing strings if possible. This ADIF scheme can be parametrized	For more information about the automated data translation capability as well as the configuration required to execute automated

Group	ADIF Import Scheme	Provided Functionality	For More Informaton See
Automated Translations		with an object reference string. This ADIF scheme can be executed for object classes for which the **Enable Data Translation** attribute is set to `True`. and that have at least one property set to `True` in the **Property Automated Translation** attribute.	translations, see the section *Configuring the Translation of Object Data* in the chapter *Localization and Multi-Language Support for the Alfabet Interface* in the reference manual *Configuring Alfabet with Alfabet Expand*.
Alfabet Instance Automated Translations	`GetAutoTranslationsForMM`	This private ADIF scheme uploads automated translations to the translation tables. Please note that to update the translations to reports and report folders, the **Update Translation** functionality available on the root node of the **Reports** tab must be triggered. To update the translations to workflows, the **Update Workflow Translation** functionality in the **Globalization** menu must be triggered.	For more information about the automated data translation capability as well as the configuration required to execute automated translations, see the section *Configuring the Translation of Object Data* in the chapter *Localization and Multi-Language Support for the Alfabet Interface* in the reference manual *Configuring Alfabet with Alfabet Expand*.
Alfabet Instance Automated Translations	`Import_Instance_Validated_Auto_Texts`	This protected ADIF scheme is the complementary to the ADIF job `Export_Instance_Automated_Texts` and is used to upload translated strings that have been reviewed and corrected. This ADIF scheme can be executed for object classes for which the **Enable Data Translation** attribute is set to `True`. and that have at least one property set to `True` in the **Property Automated Translation** attribute.	For more information about the automated data translation capability as well as the configuration required to execute automated translations, see the section *Configuring the Translation of Object Data* in the chapter *Localization and Multi-Language Support for the Alfabet Interface* in the reference manual *Configuring Alfabet with Alfabet Expand*.
Alfabet Instance Automated Translations	`Renew_Instance_Automated_Translations_Automated_Texts`	This protected ADIF scheme refreshes automated translations for permissible object class properties reusing existing strings if possible. This ADIF scheme can be executed for object classes for which the **Enable Data Translation** attribute is set to `True`. and that have at least one property	For more information about the automated data translation capability as well as the configuration required to execute automated translations, see the section *Configuring the Translation of Object Data* in the chapter *Localization and Multi-Language Support for the Alfabet*

Group	ADIF Import Scheme	Provided Functionality	For More Informaton See
		set to `True` in the **Property Automated Translation** attribute.	*Interface* in the reference manual *Configuring Alfabet with Alfabet Expand*.
Alfabet Instance Automated Translations	`Rescan_Instance_Translations_To_Tmp`	This protected ADIF scheme extracts existing manual translations to the internal translation table `ALFA_INSTANCE_VOCABULARY` which will be used to match newly saved texts to existing translations thus providing for reuse of existing translations. It is recommended that customers run this ADIF job when they begin using the automated data translation capability. This ADIF scheme can be executed for object classes for which the **Enable Data Translation** attribute is set to `True`. and that have at least one property set to `True` in the **Property Automated Translation** attribute.	For more information about the automated data translation capability as well as the configuration required to execute automated translations, see the section *Configuring the Translation of Object Data* in the chapter *Localization and Multi-Language Support for the Alfabet Interface* in the reference manual *Configuring Alfabet with Alfabet Expand*.

The following information is available:

- [Import Scheme to Automatically Reallocate Cost Centre Costs](#)

  - [Use Case](#)

  - [Understanding the CC_Reallocate Scheme](#)

  - [Adapting the Scheme to the Current Environment Prior to Use](#)

    - [Selecting a Subset of the Available Cost Centres for Update](#)

    - [Taking Monetary IDs Other Than Current Into Account](#)

Import Scheme to Automatically Reallocate Cost Centre Costs

Scheme Type: Import Scheme

Name : **CC_Reallocate**

Supported Alfabet Functionality: **Cost Management**

Provided Functionality: Reallocation of costs to objects assigned to cost centres after cost assignment to cost centres has changed.

Use Case

The **Cost Management** functionality in Alfabet allows you to manage costs based on cost centres. The cost centre is a means to centrally define costs and allocate them according to a defined allocation scheme among a group of architecture objects. In a cost centre, you define the costs of the cost centre for different years and cost types, the objects to which the costs are allocated, and the way that costs are allocated to the objects.

When the costs assigned to the cost centre are changed on the **Cost Accrual** page view of the cost centre, the cost allocation to objects is updated automatically. Assignment of new objects to the cost centre or manual definition of cost distribution rates on the **Objects** page view of the cost centre requires manual update of the cost allocation. The manual update is performed by clicking the **Reallocate Costs** button on the **Objects** page view.

> (i) Cost centres are configured in the **Cost Centres** functionality in Alfabet. For more information about cost centres, see the section *Configuring Cost Centres for Cost Management Capabilities* in the reference manual *Configuring Evaluation and Reference Data in Alfabet*.

ADIF can be used to batch update budget costs and object assignment of a cost centre as well as cost allocation rates. For all imports via ADIF, cost allocation is not performed automatically but must be updated manually. This can be done in two ways:

- In the Alfabet interface, navigate to the **Objects** page view of each cost centre and click the **Reallocate Costs** button.

or

- Execute the preconfigured ADIF scheme **CC_Reallocate** via the ADIF console application to batch update all cost allocations for all cost centres. For information about how to run an ADIF scheme job, see *Starting Data Import, Export, or Manipulation via ADIF*.

Understanding the CC_Reallocate Scheme

This section describes the processing done via the **CC_Reallocate** scheme. The information is not required to run the scheme. You can read this information if you want to import cost centre related data via ADIF and are not familiar with the underlying database object structure yet.

It is important to understand the way that changes to a cost centre configuration are processed in Alfabet to understand the **CC_Reallocate** Scheme.

The following Alfabet object classes are involved in the process:

Name	Caption	Description
CostCentre	Cost Centre	Representation of a cost centre object.
CostCentreAllocation	Cost Centre Allocation	Supporting class that stores information about object assignment to cost centres.
BudgetValue	Budget Value	Supporting class that stores information about cost values in the cost management functionality.

When an object is assigned to the cost centre on the **Objects** page view of the cost centre, a new instance of the object class CostCentreAllocation is created that references both the cost centre and the object. In addition, it stores the percentage of the cost centre costs that shall be assigned to an object if the cost allocation (a percentage) is manually defined.
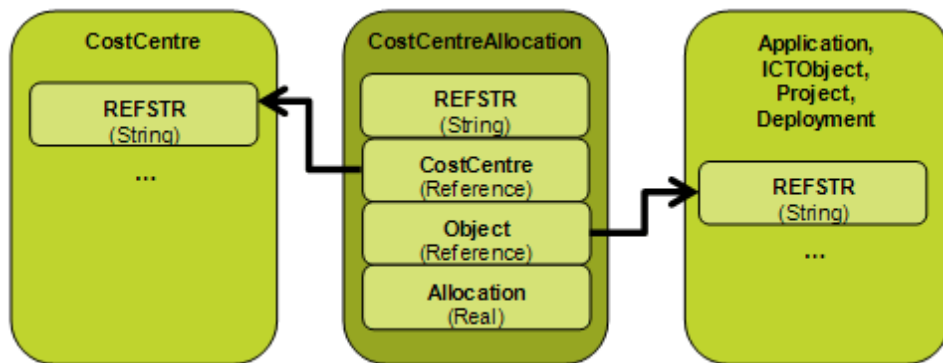
Please note the following:

- Cost centres are created in the **Cost Centres** functionality in Alfabet. For more information about cost centres, see the section *Configuring Cost Centres for Cost Management Capabilities* in the reference manual *Configuring Evaluation and Reference Data in Alfabet*.

- The editability of costs centres is configured in the XML object **CostManagerDef**. For more information about these configuration requirements for cost centres, see the section *Configuring the Editability of Costs in Cost Centres* in the reference manual *Configuring Alfabet with Alfabet Expand*.

- Currencies are configured in the **Reference Data** functionality in Alfabet. For more information about configuring currencies, see the section *Configuring Currencies and Currency Exchange Rates for Cost Management Capabilities* in the reference manual *Configuring Evaluation and Reference Data in Alfabet*.

CostType
REFSTR
(String)

BudgetValue
REFSTR
(String)
MonetaryType
(Reference)
Value
(Real)
Year
(Integer)
Owner
(Reference)
MonetaryUnit
(String)
MonetaryID
(String)

CostCentre
REFSTR
(String)
...

CostCentreAllocation
REFSTR
(String)
CostCentre
(Reference)
Object
(Reference)
Allocation
(Real)

Application,
ICTObject,
Project,
Deployment
REFSTR
(String)
...
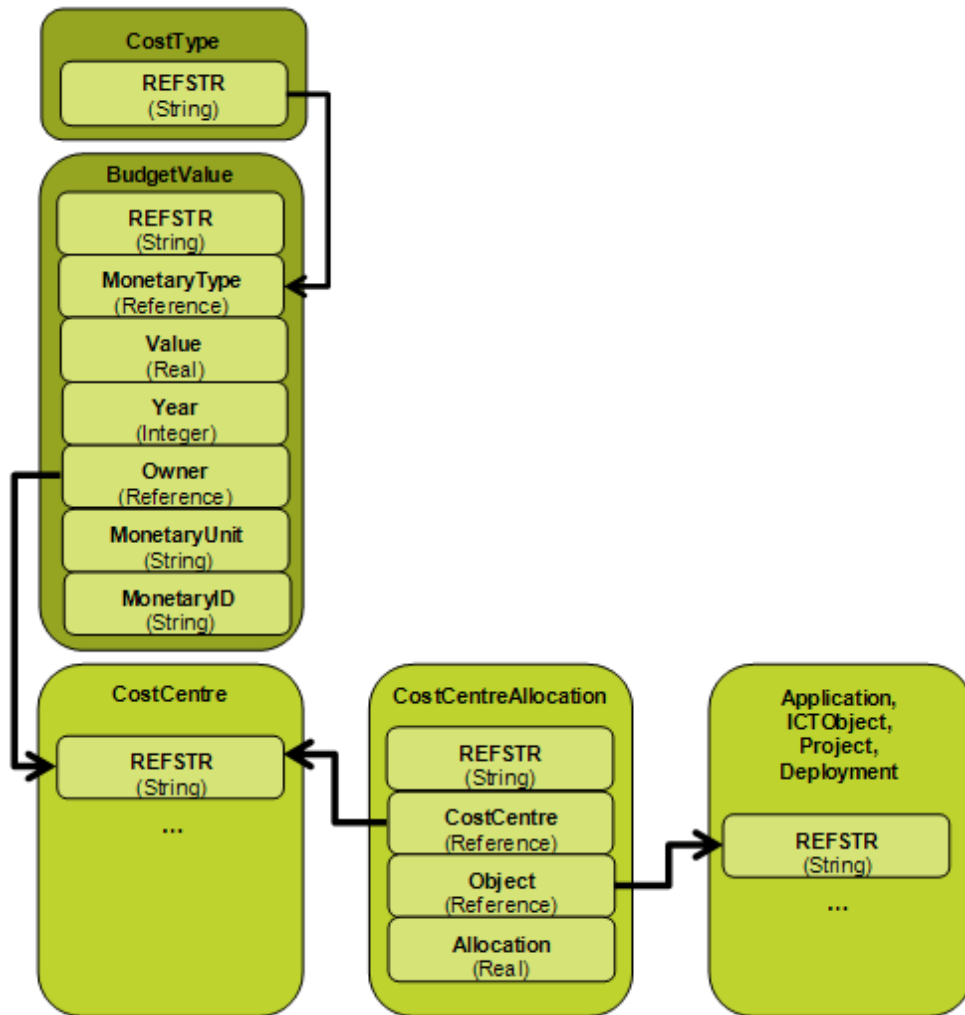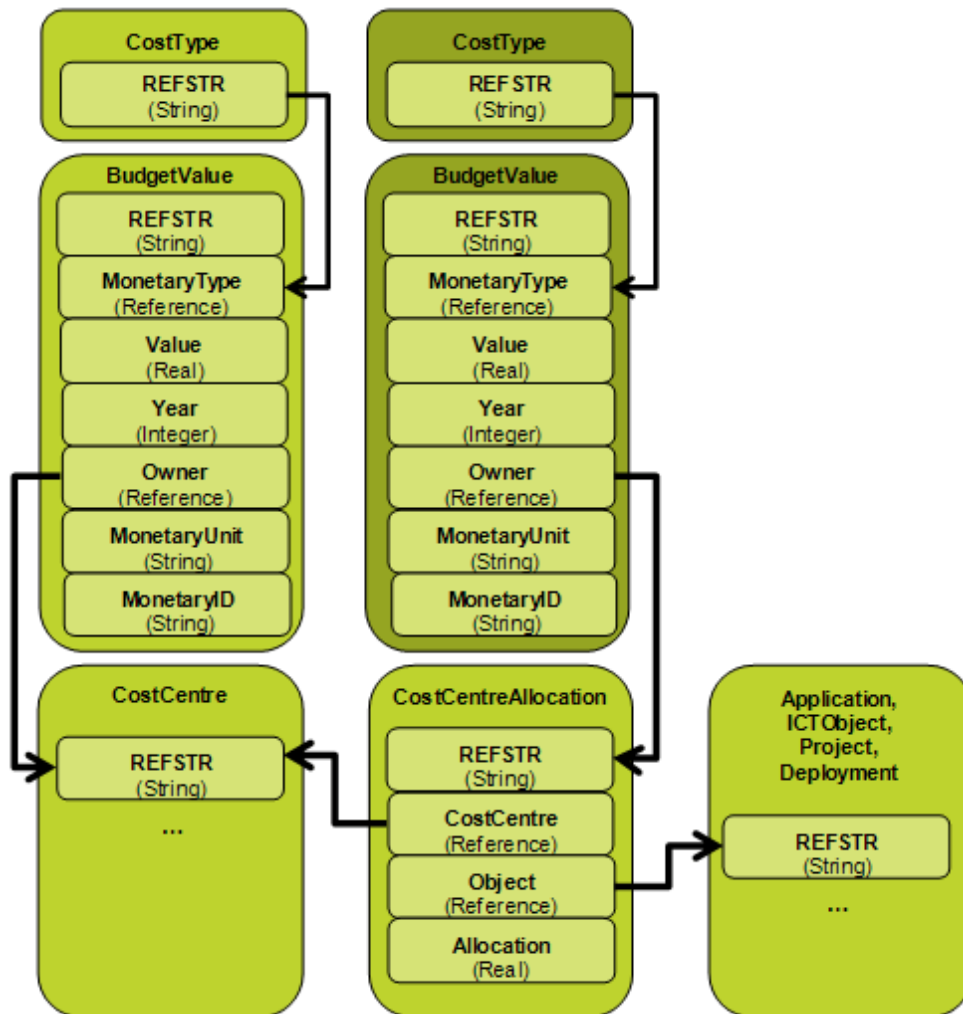
In a second step, the budget assigned to the object is updated. The object's budget is also defined in instances of the object class `BudgetValue`.

If an object is only assigned to one cost centre, the budget value instances assigned to the object are identical to the budget value instances of the `CostCentreAllocation` referencing the object, except that the `Owner` is the object instead of the `CostCentreAllocation`.

If an object is assigned to multiple cost centres, the budget value instances of the object are the sum of all budget values where the owner is a `CostCentreAllocation` with a reference to the object and an identical specification of the year, cost type and monetary ID.
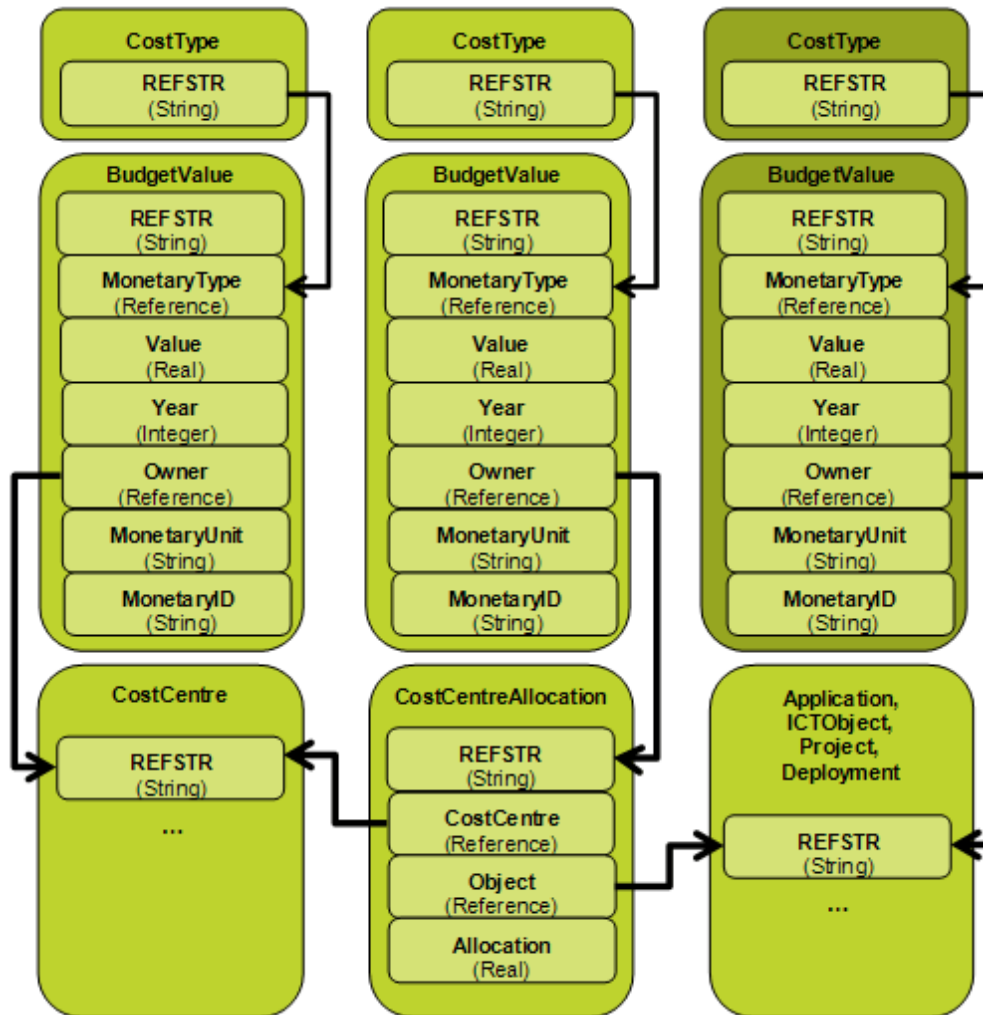
For example, a cost centre has two objects assigned: application 1 and application 2. The cost allocation is defined manually. 60 % of the costs are assigned to application 1 and 40 % of the costs are assigned to application 2. This information is stored in two `CostCentreAllocation` instances in the Alfabet database.

The costs are defined for the years 2010 and 2011 and for the cost types Maintenance and License. The monetary ID and the monetary unit are identical for all cost definitions and are not included in the example in order to keep it simple.

The budget specification on the cost centre leads to the generation of four instances of the object class `BudgetValue`, one for each combination of year and cost type.

During cost allocation, `BudgetValue` instances are created for the two `CostCentreAllocation` instances of the cost centre. Each `CostCentreAllocation` has the same amount of `BudgetValue` instances assigned as the cost centre. The `BudgetValue` instances are identical to the `BudgetValue` instances of the cost centre except that the Owner is the `CostCentreAllocation` and the Value is the percentage of the cost value for the cost centre defined by the cost allocation rate.

Each object assigned to the cost centre allocation then gets the same `BudgetValue` instances defined with the owner set to the object. Application 1 only gets cost allocated from Cost Centre 1. Therefore the Value settings of the `BudgetValue` instances of the object are identical to the Value settings of the `BudgetValue` instances of `CostCentreAllocation` 1. The licence fees for application 2 are allocated via two different cost centres. Therefore, the Value defined in the `BudgetValue` for application 1 is the sum of the `BudgetValue` instances from the cost centre 1 and the other cost centre.

Application 1 ←→ CostCentreAllocation 1 / Allocation=60% ←→ Cost Center 1 ←→ CostCentreAllocation 2 / Allocation=40% → Application 2

Application 1	CostCentreAllocation 1	Cost Center 1	CostCentreAllocation 2	Application 2	
BudgetValue Year=2010 Value=1200 CostType=Maintenance Owner=Application1	BudgetValue Year=2010 Value=1200 CostType=Maintenance Owner=CCAllocation1	BudgetValue Year=2010 Value=2000 CostType=Maintenance Owner=CostCenter1	BudgetValue Year=2010 Value=800 CostType=Maintenance Owner=CCAllocation2	BudgetValue Year=2010 Value=800 CostType=Maintenance Owner=Application2	
BudgetValue Year=2011 Value=1320 CostType=Maintenance Owner=Application1	BudgetValue Year=2011 Value=1320 CostType=Maintenance Owner=CCAllocation1	BudgetValue Year=2011 Value=2200 CostType=Maintenance Owner=CostCenter1	BudgetValue Year=2011 Value=880 CostType=Maintenance Owner=CCAllocation2	BudgetValue Year=2011 Value=880 CostType=Maintenance Owner=Application2	
BudgetValue Year=2010 Value=720 CostType=Licence Owner=Aplication1	BudgetValue Year=2010 Value=720 CostType=Licence Owner=CCAllocation1	BudgetValue Year=2010 Value=1200 CostType=Licence Owner=CostCenter1	BudgetValue Year=2010 Value=480 CostType=Licence Owner=CCAllocation2	BudgetValue Year=2010 Value=480 CostType=Licence Owner=Application2	
BudgetValue Year=2011 Value=120 CostType=License Owner=Application1	BudgetValue Year=2011 Value=120 CostType=License Owner=CCAllocation1	BudgetValue Year=2011 Value=200 CostType=License Owner=CostCenter1	BudgetValue Year=2011 Value=80 CostType=License Owner=CCAllocation2	BudgetValue Year=2011 Value=110 CostType=License Owner=Application2	BudgetValue Year=2011 Value=30 CostType=License Owner=CCAllocation5

(Allocation factors shown between columns: 60% on the left side, 40% on the right side.)

The ADIF scheme CC_Reallocate performs the re-allocation of costs after a change to the configuration of cost centres has been performed via ADIF in a five step process, each being triggered by an import entry. The current information about available cost centres as well as CostCentreAllocation instances and Budget-Value instances referencing the cost centres is taken over from the Alfabet database and processed in temporary tables.

After executing the ADIF scheme CC_Reallocate, the information in the BudgetValue table in the Alfabet database is updated. The update is limited to BudgetValue instances of CostCentreAllocation instances and objects that are created via cost allocation processes.

The table provides an overview of the sequence of action performed when running the CC_Reallocate scheme:

Import Entry	Action Performed
`Select Updated Cost Centres`	Information about all `CostCentre` instances in the Alfabet database is written to a temporary table `CC_Temp`. **NOTE**: By default, data about all available cost centres is written to the temporary table. You can alter the import entry to consider only a subset of the available cost centres. For more information, see [Selecting a Subset of the Available Cost Centres for Update](#)
`Update Allocation Percentages for all CCA Objects`	Information about all `CostCentreAllocation` instances in the Alfabet database that are relevant for the cost centres in the table `CC_Temp` is written to a temporary table `CCA_Temp` together with the information about the overall number of objects assigned to a cost centre. The allocation rate is either taken over from the property `Allocation` of the `CostCentreAllocation` instances or calculated from the overall number of objects when equal allocation is activated.
`Get Related Monetary Types`	A temporary table `CCOBJ_MTYPE_TMP` is created that lists the cost types that are relevant for the cost centres. The information is required in step five to identify which parts of the cost planning of applications, projects, deployments, ICT objects, or service products are defined via cost centre allocation.
`Update CCA Budget Values`	Based on the information in the previously created temporary tables and the `BudgetValue` information in the Alfabet database, the `BudgetValue` instances for all updated `CostCentreAllocation` instances in the table CCA_Temp is calculated and written to a temporary table `MV_CCA_Temp`. The data in the temporary table is then used to update the data in the `BudgetValue` table of the Alfabet database. New instances are added, existing instances are updated or, if they are obsolete, deleted.
`Update Objects Budget Values`	Based on the information in the previously created temporary tables, the `BudgetValue` instances for all objects assigned to one or multiple cost centres in `CC_Temp` is calculated and written to a temporary table `MV_Temp`. The data in the temporary table is then used to update the data in the `BudgetValue` table of the Alfabet database. New instances are added, existing instances are updated or, if they are obsolete, deleted. **NOTE** : The calculation of budget cost values is limited to budget value instances that have a `MonetaryID` set to `Current`, because only current costs are handled via cost centres in the default configuration of Alfabet. If you altered the configuration of the cost

Import Entry	Action Performed
	definition type editability in the XML object `CostManagerDef`, you must adapt the `CC_Reallocate` scheme accordingly. For more information, see Taking Monetary IDs Other Than Current Into Account.

Adapting the Scheme to the Current Environment Prior to Use

The default update must be adapted to suit your demands. You must alter the following part of the process:

- Selecting a Subset of the Available Cost Centres for Update

Optionally, you can alter the following part of the process:

- Taking Monetary IDs Other Than Current Into Account

Selecting a Subset of the Available Cost Centres for Update

The query in the scheme that finds the cost centre data to be written to the temporary database table includes a `WHERE` clause to re-allocate costs only for a subset of the available cost centres. The example `WHERE` clause limits update to cost centres starting with'AAA' which inhibits cost re-allocation for any cost centres in most databases. Prior to using the **CC_Reallocate** scheme, you must therefore alter the query to find cost centres matching you demands.

> ⓘ When the budget values for objects are impacted by more than one cost centre, it is recommended that you update all cost centre data without restricting the update to a subset of updated cost centres to make sure that all relevant updates are performed.

To define cost allocation via the **CC_Reallocate** scheme for a defined set of cost centres in the Alfabet database:

1) In the ADIF explorer of Alfabet Expand, go to **CC_Reallocate** > **Select Updated Cost Centres** > **SQL Commands - OnComplete** > **Insert Updated cost centres**.

2) In the attribute window of the SQL Command **Insert Updated cost centres** click the **Browse** ![button] button in the **Text** attribute field. The query of the SQL command is displayed in a text editor.

3) Alter the WHERE clause in the query to define which cost centres shall be included in the cost centre allocation process or delete the WHERE clause to update all cost centres.

> The following example displays the query with a WHERE clause restricting cost re-allocation to cost centres that are owned by the organization 'Corporate IT'. Please note that the definition of the condition requires that two objects are found by the query and therefore the SELECT clause must be altered to include class specifications. These class specification must then be removed from the resulting column names by defining an alias for the column name.

```
INSERT CC_TMP

SELECT cc.REFSTR, cc.NAME As 'NAME', cc.EQUALALLOCATION As 'EQUALALLOCATION'

FROM COSTCENTRE cc, ORGAUNIT org

WHERE cc.OWNER = org.REFSTR

    AND org.NAME = 'Corporate IT'
```

4) Click **OK** to save your changes.

Taking Monetary IDs Other Than Current Into Account

The property MonetaryID of the class BudgetValue defines, whether the budget value is defined for a cost request, budgeted costs, current costs, committed costs or forecast costs. By default, Alfabet is configured to manage only current costs via cost centres and the **CC_Reallocate** scheme therefore only handles Budget-Value instances assigned to objects that have a MonetaryID set to Current.

You can configure Alfabet to manage other cost definition types via cost centres. Configuration is carried out in the XML object **CostManagerDef** in the configuration tool Alfabet Expand.

> For more information about the configuration of the XML object **CostManagerDef**, see *Configuring Cost Management Capabilities* in the reference manual *Configuring Alfabet with Alfabet Expand*.

While most of the **CC_Reallocate** Scheme handles `BudgetValue` instances that are defined for a cost centre independent of the `MonetaryTypeID`, the query that scans the Alfabet database for `BudgetValue` instances that are obsolete uses the `MonetaryID` to define whether a `BudgetValue` instance owned by an object was created due to a cost centre cost allocation.

To include other cost definition types in the update of budget values with the **CC_Reallocate** scheme:

1) In the ADIF explorer of Alfabet Expand, go to **CC_Reallocate** > **Update Object's Budget Values** > **SQL Commands - OnComplete** > **Delete unused objects**.

2) In the attribute window of the SQL Command **Update changed budget values** click the **Browse** button in the **Text** attribute field. The query of the SQL command is displayed in a text editor.

3) Change the `WHERE` clause that restricts data processing to `BudgetValue` instances with a `MonetaryID` set to `Current` to restrict data processing to the set of `MonetaryID`s that you want to be included.

The following example displays the query with a `WHERE` clause including the `MonetaryID`s `Current` and `Budget`:

```
DELETE FROM BUDGETVALUE

WHERE (REFSTR IN

    (SELECT BV.REFSTR

    FROM BUDGETVALUE AS BV

    INNER JOIN ON BV.MONETARYTYPE = CCOBJ_MTYPE_TMP.MONETARYTYPE AND

        BV.OWNER = CCOBJ_MTYPE_TMP.OBJECT

    WHERE (BV.REFSTR NOT IN

        (SELECT DISTINCT REFSTR

        FROM MV_TMP))

        AND

        ( BV.MONETARYCODEID ='Current'

        OR BV.MONETARYCODEID ='Budget' )))
```

4) Click **OK** to save your changes.

INDEX