How-To Guide: Business Object Layer Programming

SAP® CRM 7.0

Target Audience

- System administrators
- Technology consultants

Document version: 1.0 – December 2008

THE BEST-RUN BUSINESSES RUN SAP

THE BEST-RUN BUSINESSES RUN SAP

SAP AG
Dietmar-Hopp-Allee 16
69190 Walldorf
Germany
T +49/18 05/34 34 24
F +49/18 05/34 34 20
www.sap.com

Disclaimer

Documentation in the SAP Service Marketplace

You can find this documentation at the following address:
http://service.sap.com/

Terms for Included Open Source Software

This SAP software contains also the third party open source software products listed below. Please note that for these third party products the following special terms and conditions shall apply.

1. This software was developed using ANTLR.

2. gSOAP

3. SAP License Agreement for STLport

SAP License Agreement for STLPort between
SAP Aktiengesellschaft
Systems, Applications, Products in Data Processing
Neurottstrasse 16
69190 Walldorf, Germany
(hereinafter: SAP)
and
you
(hereinafter: Customer)

a) Subject Matter of the Agreement

A) SAP grants Customer a non-exclusive, non-transferrable, royalty-free license to use the STLport.org C++ library (STLport) and its documentation without fee.

B) By downloading, using, or copying STLport or any portion thereof Customer agrees to abide by the intellectual property laws, and to all of the terms and conditions of this Agreement.

C) The Customer may distribute binaries compiled with STLport (whether original or modified) without any royalties or restrictions.

D) Customer shall maintain the following copyright and permissions notices on STLport sources and its documentation unchanged:
Copyright 2001 SAP AG

E) The Customer may distribute original or modified STLport sources, provided that:

o The conditions indicated in the above permissions notice are met;

o The following copyright notices are retained when present, and conditions provided in accompanying permission notices are met:

Copyright 1994 Hewlett-Packard Company
Copyright 1996,97 Silicon Graphics Computer Systems Inc.
Copyright 1997 Moscow Center for SPARC Technology.
Copyright 1999,2000 Boris Fomitchev
Copyright 2001 SAP AG

D) SAP and the Customer agree that the typical foreseeable extent of damage shall under no circumstances exceed EUR 5,000.
E) The Customer shall take adequate measures for the protection of data and programs, in particular by making backup copies at the minimum intervals recommended by SAP. SAP shall not be liable for the loss of data and its recovery, notwithstanding the other limitations of the present Art. 4 if this loss could have been avoided by observing this obligation.

F) The exclusion or the limitation of claims in accordance with the present Art. 4 includes claims against employees or agents of SAP.

4. Adobe Document Services

Adobe, the Adobe logo, Acrobat, PostScript, and Reader are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and / or other countries. For information on Third Party software delivered with Adobe document services and Adobe LiveCycle Designer, see SAP Note 854621.

Typographic Conventions

Type Style	Description
Example Text	Words or characters quoted from the screen. These include field names, screen titles, pushbuttons labels, menu names, menu paths, and menu options. Cross-references to other documentation
Example text	Emphasized words or phrases in body text, graphic titles, and table titles
EXAMPLE TEXT	Technical names of system objects. These include report names, program names, transaction codes, table names, and key concepts of a programming language when they are surrounded by body text, for example, SELECT and INCLUDE.
`Example text`	Output on the screen. This includes file and directory names and their paths, messages, names of variables and parameters, source text, and names of installation, upgrade and database tools.
`Example text`	Exact user entry. These are words or characters that you enter in the system exactly as they appear in the documentation.
`<Example text>`	Variable user entry. Angle brackets indicate that you replace these words and characters with appropriate entries to make entries in the system.
`EXAMPLE TEXT`	Keys on the keyboard, for example, F2 or ENTER.

Icons

Icon	Meaning
⚠	Caution
✣	Example
💡	Note
⬆	Recommendation
◇	Syntax

Additional icons are used in SAP Library documentation to help you identify different types of information at a glance. For more information, see *Help on Help → General Information Classes and Information Classes for Business Information Warehouse* on the first page of any version of *SAP Library*.

Contents

1 Introduction

The usage of the business object layer and its uniform application programming interface (API) to access business data offers considerable advantages compared to the usage of various native APIs historically available for business objects:

- The object oriented BOL API is simple, uniform and easy to use
- The built-in buffer accelerates your applications
- Your programs are decoupled from any interface changes in the underlying business object specific APIs
- Development of WEBCUIF applications is made easy, since the BOL has been designed to work hand in hand with the UI parts of the framework

It is possible to enhance the business object layer to cover business data not yet supported: After the corresponding business objects and query services have been modeled and implemented, you can use them at runtime.

1.1 BOL Versions

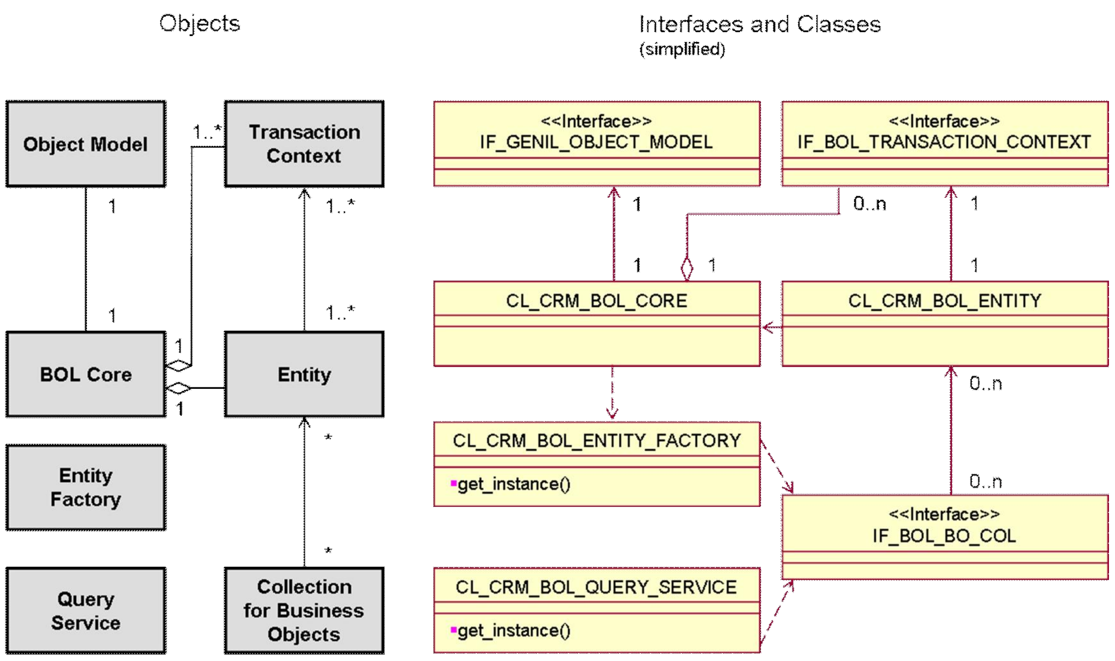
The following list shows the various versions of the Business Object Layer:

Version	Description
CRM 3.1	First version – never in productive use
CRM 4.0	First version use in a productive system
CRMIS 4.0	Several improvements – introduction of separate display mode
SAP_ABA 7.0	Relocation from CRM to ABA layer, first usage outside CRM
CRMUIF 5.1	Introduction of dynamic searches; whole CRM with BOL support
CRMUIF 5.2	Stabilization and performance improvements
CRMUIF 6.0	Major internal changes for better performance on high number of objects
WEBCUIF 7.0	Current version; Full Switch Framework support; Stable Core release

1.2 Overview

The business object layer API consists of various interfaces and classes, you use to access business data. Most important are class `CL_CRM_BOL_QUERY_SERVICE` you use to select business objects, class `CL_CRM_BOL_ENTITY` representing entities as business object segments, the interface `IF_BOL_TRANSACTION_CONTEXT` to control transactional behavior, and `IF_BOL_BO_COL` providing access to collections holding entities.

The lifetime of these objects in general is the whole session. The BOL API, however, provides a possibility to free used business objects and their memory.



Entities and related transactional contexts are centrally managed. This influences their lifetime but ensures also instance uniqueness. Therefore it is sufficient to compare object instances for equality instead of object type and key.

2 Basic Features of the BOL Application Programming Interface

When you use the business object layer to work with business objects in an ABAP application you typically use code sequences similar to those indicated in the following chapters.

2.1 Setting up a BOL Instance

You create an instance of the business object layer by calling the static factory method of its core class:

Syntax

```
* Start BOL Core module
DATA: lv_bol_core TYPE REF TO cl_crm_bol_core.
lv_bol_core = cl_crm_bol_core=>get_instance( ).
lv_bol_core->start_up( ‘MY_COMPONENT_SET’ ).
```

Since the BOL core CL_CRM_BOL_CORE follows the singleton design pattern, you can just have one instance of it. The START_UP method takes the name of the Generic Interaction Layer (GenIL) component set you want to start as input parameter and starts the BOL. From now on you can use all services of the BOL and the component set loaded.

The START_UP method takes another optional parameter IV_DISPLAY_MODE_SUPPORT, which is set to ABAP_FALSE by default. This means the BOL follows the optimistic locking approach, which makes all objects appear changeable in the user interface even without a lock (entity method IS_CHANGEABLE always return ABAP_TRUE). The lock is automatically requested on the first attempt to really make a change. The lock state of an entity could be check via method IS_LOCKED.

If the parameter IV_DISPLAY_MODE_SUPPORT is set to ABAP_TRUE the BOL follows the strict locking approach where only locked objects appear as changeable: For more information, see *Display Mode Support* [page 23].

2.2 Component Sets and Components of the Generic Interaction Layer

Each component set defines a set of GenIL components, each providing specific business objects with dependent objects and related queries. GenIL component sets and GenIL components are defined in Customizing for *Customer Relationship Management* under *CRM Cross-Application Components → Generic Interaction Layer/Object Layer → Basic Settings*.

Components of the GenIL are implemented ABAP classes. In order to find out which business objects and attributes, which relations, dependant objects, queries and further services are provided by a component or component set, you may use the so called *Model Browser* (transaction GENIL_MODEL_BROWSER).

The following code snippet shows how to load additional components or component sets and how to check the components loaded so far:

Syntax

```
* Load additional component
DATA: lv_component_name type crmt_component_name.
lv_component_name = 'ANOTHER_COMPONENT'.
```

```
lv_bol_core->load_component( lv_component_name ).

* Load additional component set

DATA: lv_component_set_name type crmt_genil_appl.

lv_component_set_name = 'ANOTHER_COMPONENT_SET'.

lv_bol_core->load_component_set( lv_component_set_name ).

* Get components loaded

DATA: lv_obj_model TYPE REF TO if_genil_obj_model,

lv_obj_model_ TYPE REF TO cl_crm_genil_obj_model,

lt_components_loaded type genil_component_tab.

lv_obj_model =
cl_crm_genil_model_service=>get_runtime_model( ).

lv_obj_model_ ?= lv_obj_model.

lt_components_loaded = lv_obj_model_->get_components_loaded(
).
```

Note

Via this technique it is also possible to work fully without predefined component sets. Just start the BOL core with predefined component set EMPTY, which does not include any component. After this you may load component by component via LOAD_COMPONENT method.

2.3 Issue Queries

Before you can work with business objects or “entities” you have to find them. That’s why you usually start your application with a search. You use a query service object to receive a collection of entities that match the search criteria given.

2.3.1 Simple Queries

In order to fire a query you need to instantiate the query service you want to use and provide it with the search criteria.

To get an instance of the generic query service CL_CRM_BOL_QUERY_SERVICE via the factory method GET_INSTANCE. Use its SET_PROPERTY method to indicate the search criteria and its GET_QUERY_RESULT method to fire your query and to retrieve a result list of entities (represented by IF_BOL_ENTITY_COL):

Syntax

```
* Create a query service

DATA: lv_query TYPE REF TO cl_crm_bol_query_service.

lv_query = cl_crm_bol_query_service=>get_instance(
‘UIFSearchConnection’ ).

* Set a search criterion

lv_query->set_property( iv_attr_name = ‘CARRID’

iv_value = ‘AA’ ).
```

```
* Read a search criterion
DATA: lv_carrid TYPE string.
lv_carrid = lv_query->get_property_as_string( ‘CARRID’ ).
* Execute query and receive result
DATA: lv_result TYPE REF TO if_bol_entity_col.
lv_result = lv_query->get_query_result( ).
```

As you can see from the example it is also possible to retrieve search criteria from the query service.

In a generic application, such as the BOL browser, you may use the BOL Object Model Service to determine available query services of the component set and components loaded.

Syntax

```
* Determine available query services
DATA: lv_obj_model TYPE REF TO if_genil_obj_model,
lt_query_names TYPE crmt_ext_obj_name_tab.
lv_obj_model =
cl_crm_genil_model_service=>get_runtime_model( ).
lv_obj_model->get_object_list(
EXPORTING iv_object_kind = if_genil_obj_model=>query_object
IMPORTING ev_object_list = lt_query_names ).
```

You can perform queries and many other methods of the BOL API using the so called BOL Browser (transaction GENIL_BOL_BROWSER).

1. Select the component or component set providing the objects you are interested in, e. g. component set SAMPLE containing sample orders, the OrderQuery, etc.
2. Select the query and enter search criteria before you launch the query.
3. To see a particular object with it attributes, double-click the object ID displayed in the *List Browser*.

2.3.2 Dynamic Queries

As of SAP CRM release CRMUIF 5.1 the BOL supports so called dynamic queries provided with enhanced features:

- In addition to the EQUAL operator, you can use arbitrary operators like GREATER THAN, LESS THAN, CONTAINS, NOT CONTAINS, etc. to specify the search criteria
- You can search for multiple values for one search criterion at the same time (logical OR)
- You can save and retrieve predefined searches with preset search criteria as templates with an arbitrary name

Syntax

```
* Get advanced query
DATA: lv_dyn_query TYPE REF TO cl_crm_bol_dquery_service.
lv_dyn_query =
cl_crm_bol_dquery_service=>get_instance(
'UIFAdvSearchFlight' ).
```

```
* Set general query parameter for maximum number of hits
lv_dyn_query->set_property( iv_attr_name = 'MAX_HITS'
iv_value = '5' ).
* Add selection criteria: Maximum Seats > 5
lv_dyn_query->add_selection_param( iv_attr_name = 'SEATSMAX'
                             iv_sign = 'I'
                             iv_option = 'GT' “Greater than
                             iv_low = '100'
                             iv_high = '' ).
* Execute the query and receive result
DATA: lv_result type ref to if_bol_entity_col.
lv_result = lv_advanced_query->get_query_result( ).
```

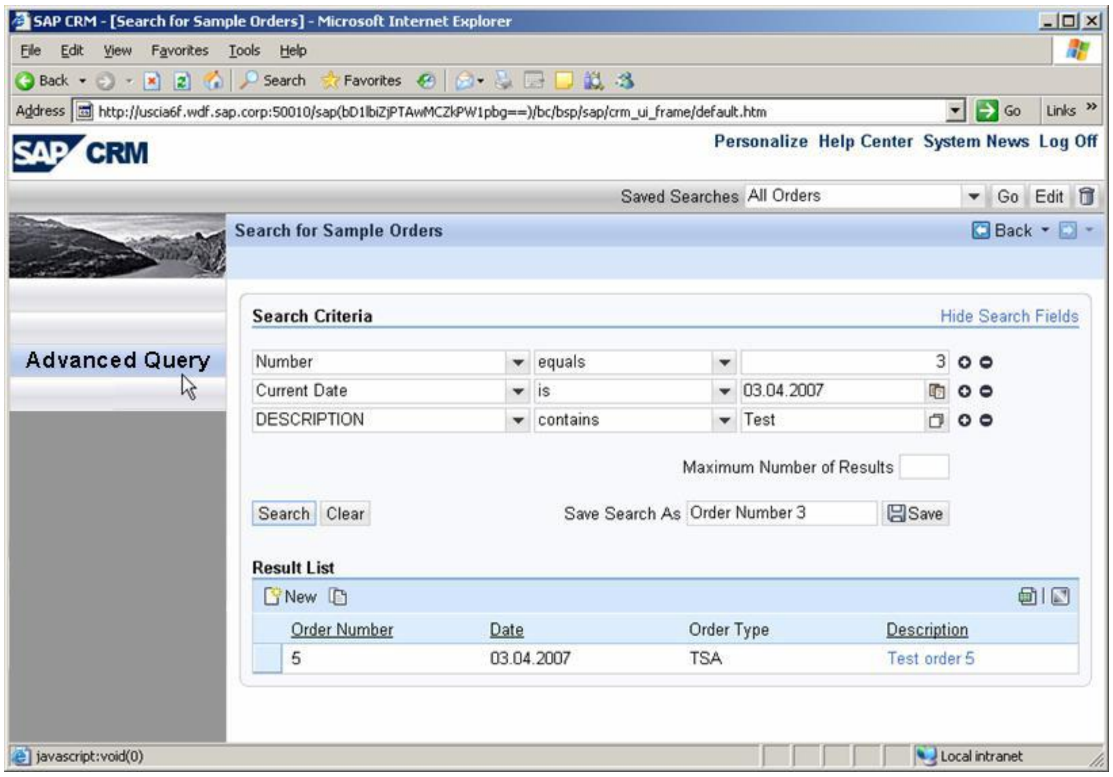
Because of the dynamic nature the selection parameters of a dynamic query are organized in a collection. You can access them via method GET_SELECTION_PARAMS, which returns directly the collection the dynamic query service itself operates on. You can iterate on this collection in order to check or change all selection parameters.

Besides adding of selection parameters, it is also possible to insert them. In a normal program this makes not really sense. However you high directly display the parameter collection in a UI where you normally want to group the criteria by the parameter name.

Syntax

```
* Save query as template
lv_dyn_query->save_query_as_template( iv_query_id = ‘My
Query’
iv_overwrite = abap_true ).
...
* Load query from template
lv_dyn_query->load_query_template( iv_query_id = ‘My Query’
).
...
```

Special easy-to-use tags have been developed to display the enhanced features of dynamic queries in the Web UI:


SAP CRM - [Search for Sample Orders] - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://uscia6f.wdf.sap.corp:50010/sap(bD1lbiZjPTAwMCZkPW1pbg==)/bc/bsp/sap/crm_ui_frame/default.htm
Go Links
SAP CRM
Personalize Help Center System News Log Off
Saved Searches All Orders Go Edit
Search for Sample Orders
Back
Advanced Query
javascript:void(0)
Local intranet

Search Criteria — Hide Search Fields

Number	equals	3
Current Date	is	03.04.2007
DESCRIPTION	contains	Test

Maximum Number of Results

Search Clear — Save Search As: Order Number 3 — Save

Result List

New

Order Number	Date	Order Type	Description
5	03.04.2007	TSA	Test order 5

2.4 Read Access to Business Objects Using Entities

After you get a list of entities from a query service, you can use them in your application by displaying their attributes on the user interface, modifying them, deleting them, etc.

2.4.1 Access Attributes

The following coding example shows how to access entities of the query result and how to read their attributes.

Note

The methods are the same for all kinds of business objects.

Syntax

```
* Use iterator to access entities in query result
DATA:lv_iterator TYPE REF TO if_bol_entity_col_iterator.
lv_iterator = lv_result->get_iterator( ).
DATA: lv_entity TYPE REF TO cl_crm_bol_entity.
lv_entity = lv_iterator->get_first( ). “Entity is UIFFlight
here
WHILE lv_entity IS BOUND.
```

```
* Access attributes of business objects selected
DATA: lv_carrid TYPE ref to S_CARR_ID,
lv_connid TYPE S_CONN_ID,
lv_flightdate TYPE string.
lv_carrid ?= lv_entity->get_property( ‘CARRID’ ).
lv_entity->get_property_as_value( EXPORTING iv_attr_name =
‘CONNID’
IMPORTING ev_result = lv_connid ).
lv_flightdate = lv_entity->get_property_as_string( ‘FLDATE’
).
* Get all attributes of an entity
DATA: ls_attributes TYPE CRMS_BOL_UIF_TRAVEL_FLIGHT_ATT.
lv_entity->get_properties( IMPORTING es_attributes =
ls_attributes ).
*
lv_entity = lv_iterator->get_next( ).
ENDWHILE.
```

You can access attributes in three different formats:

- Method GET_PROPERTY returns a pointer to the actual value.
- Method GET_PROPERTY_AS_VALUE provides the actual value as exporting parameter.
- Method GET_PROPERTY_AS_STRING returns the value converted into and string.

Note

This conversion considers some display formatting rules but not all. So this resulting string is not intended for direct use in a business application UI.

The attribute access is covered by generic interface IF_BOL_BO_PROPERTY_ACCESS. This interface is provided not only from entities but also from query services to allow a uniform attribute access. For more information, see *Standard Interface to Access Properties of Business Objects* [page 23].

2.4.2 Access to Related Entities

The following coding example shows how to navigate from one entity to another. This navigation follows the relations given in the object model.

Syntax

```
* Get a list of 1:N related entities
DATA: lv_bookings TYPE REF TO if_bol_entity_col.
lv_bookings = lv_flight->get_related_entities(
iv_relation_name = ‘FlightBookRel’ ).
DATA: lv_booking TYPE REF TO cl_crm_bol_entity.
lv_booking = lv_bookings->get_first( ).
* Get a 1:1 related entity
```

```
DATA: lv_customer TYPE REF TO cl_crm_bol_entity.
lv_customer = lv_booking->get_related_entity(
'BookCustomerRel' ).
* Get back to the parent
lv_entity = lv_booking->get_parent( ).
```

In general, we distinguish between 1:N and 1:1 relations. Method GET_RELATED_ENTITIES works for both relations and returns always a collection. For convenience, method GET_RELATED_ENTITY returns directly the related entity for 1:1 relations.

To return the parent object, generic method `GET_PARENT` is provided.

Note

Since the parent is not necessarily unique it would return the last used. Also, the parent can only be returned for non-root objects. So if you follow an association to another root entity you cannot return via GET_PARENT.

Furthermore, it is possible to get to the root entity of a business object via method GET_ROOT.

2.4.3 Further Operations on Entities

In addition to the access to attributes and the navigation, an entity provides further functionalities:

- You can check the changeability with methods IS_CHANGEABLE and IS_LOCKED. For more information, see *Display Mode Support* [page 23].
- You may check if an entity instance is still valid, which means that it represents existing data, via method ALIVE.
- You can directly access model data, like the entity name with GET_NAME or the type of the attribute structure with GET_ATTR_STRUCT_NAME.
- You can check the properties of an attribute if it is changeable, hidden, mandatory, etc. with method GET_PROPERTY_MODIFIER. For more information, see *Advanced Entity Features* [page 37].

Further functions are described in the following sections.

2.5 Changing Business Objects Using Entities

2.5.1 Transaction Contexts

One of the most important aspects of BOL programming is to know how to modify data. You can create, modify, and delete entities in accordance to the BOL transaction model, which supports several kinds of so called transaction contexts to handle entity operations consistently.

The global transaction context holds all modified root entities, whereas the single business object scope transaction context exists for each root entity instance. The custom transaction context can be used to handle special situations, where more than one but not all modified objects belong to a transaction. In this section we only use the global context. For more information on the possible scopes of transaction contexts, see *Transaction Handling* [page 35].

The global transaction context always exists and is accessible via the BOL core instance. A single BO transaction context (fine granular context) is created either on request, via method GET_TRANSACTION on an entity, or latest when the object gets locked. Each single BO transaction context has to be finished once it was created. This happens either by calling methods SAVE and COMMIT or method REVERT on the context or via a higher order transaction context (global or customer). Additionally, the methods CHECK_SAVE_NEEDED and CHECK_SAVE_POSSIBLE allow checking before saving a BO transaction context if saving is required and possible.

Note

Always use methods CHECK_SAVE_NEEDED and CHECK_SAVE_POSSIBLE. Once saving has failed there is no way back.

Once the transaction has been saved it has to be finished with methods COMMIT or ROLLBACK.

In general, the end of a transaction context also removes the pending locks from the covert objects. If only an intermediate save is required the `COMMIT` method allows keeping the locks via optional parameter `IV_KEEP_LOCKS` set to `ABAP_TRUE`.

The REVERT method always removes the lock. All changes covered by the transaction will be rejected and the object returns to the last saved state of the database. Normally this synchronization happens directly in the REVERT method. However, for performance reasons it might be better to do this synchronization in a lazy mode. To do so the optional input parameter `IV_SUPPRESS_BUFFER_SYNC` needs to be set to `ABAP_TRUE`.

For more information, see *Transaction Handling* [page 35].

2.5.2 Locking Entities

In the default optimistic lock mode (see *Setting up a BOL Instance* [page 11]), you should lock an entity before you modify it. However, the entity‘s set-methods check if the entity is locked and, if not, try to do this. The setter only modifies properties in case the entity is locked. With display mode support switched on, locking is mandatory.

The current locking granularity of the BOL are the root object instances. So the lock request for an entity is always delegated to the corresponding root instance.

The following code fragment shows how to lock an entity.

Syntax

```
* Lock BOL entity
DATA: lv_success TYPE abap_bool.
lv_success = lv_entity->lock( ).
```

In case the lock was set the return value `LV_SUCCESS` is true (`ABAP_TRUE`). The current lock state of an entity can be checked using method IS_LOCKED. With setting a lock successfully, a transaction context is created for the corresponding root entity instance. The only way to unlock the entity is to finish this transaction context.

2.5.3 Modifying Entity Attributes

Now we want to modify properties of entities. This is simply done by using the set-methods of the entity. If you have not started a transaction before modifying properties it will be created automatically. The following code fragment shows how to modify a property of a booking.

Properties of entities can simply be modified by using the set-methods of the entity.

Syntax

```
* 1. Lock an entity and modify a property
* here booking entity with technical name ‘UIFBokking’
```

```
lv_booking = lv_flight->get_related_entity(
‘FlightBookingRel’ ).
IF lv_booking->lock( ) = ABAP_TRUE.
   lv_booking->set_property( iv_attr_name = ‘SMOKER’
                             iv_value = ‘X’ ).
ENDIF.
* 2. send all changes to BO layer
lv_bol_core->modify( ).
* 3. get the implicitly created global transaction
DATA: lv_transaction TYPE REF TO if_bol_transaction_context.
lv_transaction = lv_bol_core->get_transaction( ).
* 4. save and commit your changes
lv_transaction->save( ).
lv_transaction->commit( ).
```

Step 1 modifies the properties of an entity and step 2 communicates all modifications done so far to the Application Programming Interface (API) layer. Step 3 provides access to the automatically created global transaction context. Finally, the changes are saved and committed in step 4.

The given example works with or without display mode support, since the lock is explicitly set.

Note

Independent of the lock state you can only modify changeable attributes. The actual changeability of an attribute could be checked with method IS_PROPERTY_READONLY.

2.5.4 Creating Entities

The following code example shows how to create a root entity together with two related entities.

Syntax

```
* 1. Build parameters to create an entity:
* here a flight entity with technical name ‘UIFFlight’
DATA: lt_params TYPE crmt_name_value_pair_tab,
ls_params TYPE crmt_name_value_pair.
ls_params-name = ‘CARRID’.
ls_params-value = ‘AC’.
APPEND ls_params TO lt_params.
* 2. Get factory for business object
DATA: lv_flight_factory TYPE REF TO
cl_crm_bol_entity_factory.
lv_flight_factory = lv_bol_core->get_entity_factory(
‘UIFFlight’ ).
* 3. Create root entity
```

```
DATA: lv_flight TYPE REF TO cl_crm_bol_entity.
lv_flight = lv_flight_factory->create( lt_params ).
* 4. Create child objects
DATA: lv_booking TYPE REF TO cl_crm_bol_entity,
lv_ticket TYPE REF TO cl_crm_bol_entity.
lv_booking = lv_flight->create_related_entity(
'FlightBookRel' ).
lv_booking->set_property( iv_attr_name = 'CARRID'
iv_attr_value = 'AC' ).
lv_ticket = lv_booking->create_related_entity(
'BookTicketRel' ).
lv_ticket->set_property( iv_attr_name = 'CARRID'
iv_attr_value = 'AC' ).
* 5. Submit child objects created
lv_bol_core->modify( ).
* 6. Save and commit changes using single object transaction
context
DATA: lv_transaction TYPE REF TO if_bol_transaction_context
lv_transaction = lv_flight->get_transaction( ).
lv_transaction->save( ).
lv_transaction->commit( ).
```

It is important to distinguish between the creation of root objects via the entity factory and the creation of dependent or child objects with the CREATE_RELATED_ENTITY method. The creation of root objects directly triggers a call of the underlying API. Depending on the model, the creation of dependent or child objects normally does not trigger the API call. Therefore, it is necessary to explicitly trigger the API call by using the MODIFY method of the BOL core, which sends changes to the underlying generic interaction layer. Without this call, the created child objects will only exist in the BOL buffer and never get saved. The separate MODIFY call is used to collect fine granular changes and to bundle them before sending them to the API to process and evaluate them together. For options to control what is send by the MODIFY method, see *Excluding Entities from the Central Modify* [page 38].

Note

For performance reasons, a newly created entity always has the automatic sending option deactivated to prevent sending empty objects. To send the new entity using the MODIFY method you need either to set at least one attribute or to activate the sending option explicitly.

Even after sending a new entity to the API it is not yet persistent. It is only persistent after finishing the transaction. The persistence state of an entity can be checked via the IS_PERSISTENT method.

2.5.5 Deleting Entities

When deleting entities the distinction between root objects and dependent or child objects is of the same importance as when creating entities.

For root object instances the call of the DELETE method is directly sent to the API where the complete aggregation hierarchy is deleted. The deletion is written to the database with an internal `COMMIT WORK`.

Syntax

```
* Delete root entity
lv_flight->delete( ).
```

When deleting a child object the deletion is not automatically sent to the API. You have to trigger this explicitly by calling the MODIFY method of the BOL core.

Syntax

```
* Delete child object
lv_booking->delete( ).
lv_bol_core->modify( ).
DATA: lv_transaction TYPE REF TO if_bol_transaction_context.
lv_transaction = lv_bol_core->get_transaction( ).
lv_transaction->save( ).
lv_transaction->commit( ).
```

The example code shows that it is necessary to save and commit the transaction to ensure the persistence of the deletion.

Immediately after the execution of the CL_CRM_BOL_CORE->MODIFY method and the confirmation by the underlying API the entity is deleted from the BOL buffer. The entity instance publishes this by raising the DELETED event. Any further access to the entity instance after it has been deleted can lead to a CX_BOL_EXCEPTION exception.

2.5.6 Execution of Entity Methods

To perform special business functions it is possible to call special methods on an entity that are designed and implemented for a particular object type. You can use two different methods for entities. Each of them can have an arbitrary set of import parameters. The EXECUTE method can return an entity collection of result objects. With the CRMUIF 6.0 release method type EXECUTE2 was introduced. It returns an arbitrary DDIC type instead of a collection.

The following coding example shows the usage.

Syntax

```
* Execute entity method
DATA: lv_items TYPE REF TO cl_crm_bol_entity.
lv_items->execute( iv_method_name = ‘RenumberItems’ ).
* ... with input parameters and a list of BOL entities
returned
DATA: ls_param TYPE crmt_name_value_pair,
      lt_param TYPE crmt_name_value_pair_tab,
      lv_result TYPE REF TO if_bol_entity_col.
ls_param-name = ‘PROCESS_TYPE’.
ls_param-value = ‘TSRV’.
```

```
append ls_param to lt_param.
TRY.
   lv_result = lv_order_header->execute( iv_method_name =
‘createFollowUp’
   it_param = lt_param ).
* Error handling
CATCH CX_CRM_BOL_METH_EXEC_FAILED.
* An exception is received if method has indicated an error
* and has not returned more than one entity.
   ...
ENDTRY.
```

Note

Before an entity method is processed all pending changes in the BOL buffer are transferred automatically to the API with an implicit MODIFY.

Entity methods can also include changes of a business object. These changes lead to automatic locking and the creation of an associated transaction context.

3 Advanced Features of the BOL Application Programming Interface

3.1 Display Mode Support

With SAP_ABA 7.0 the Business Object Layer optionally supports the display mode for entities. It is activated by default if the BOL core is started with the parameter `IV_DISPLAY_MODE_SUPPORT = ABAP_TRUE`.

In display mode, a call of the IF_BOL_BO_PROPERTY_ACCESS~IS_PROPERTY_READONLY method on an entity returns ABAP_TRUE for each property. Any attempt to change properties, to create or to delete dependent objects is ignored. Such attempts can be monitored via checkpoint group BOL_ASSERTS under sub key READ-ONLY VIOLATION.

You can only change entities in the change mode. You can go to change mode by calling the SWITCH_TO_CHANGE_MODE or the LOCK method. If no lock can be obtained, the entity will remain in display mode. The current state of an entity can be checked with the IS_CHANGEABLE method, which is with display mode support equal to method IS_LOCKED.

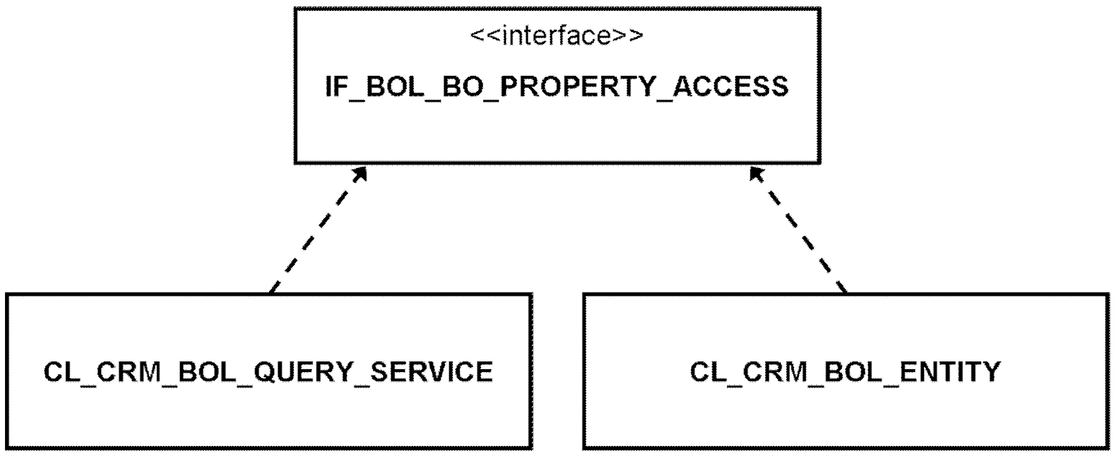
Syntax

```
* Start BOL core with display mode support
DATA: lv_bol_core TYPE REF TO cl_crm_bol_core.
lv_bol_core = cl_crm_bol_core=>get_instance( ).
lv_bol_core->start_up( iv_appl_name = 'MY_COMPONENT_SET'
                     iv_display_mode_support = ABAP_TRUE ).
* Execute query and access entity
DATA: lv_entity type ref to cl_crm_bol_entity.
lv_entity = ... "Entity is in now display mode
* Switch to change mode and change entity
lv_entity->switch_to_change_mode( ). "Reread and lock entity
IF lv_entity->is_changeable( ) = abap_true.
   lv_entity->set_property_as_string( iv_attr_name =
'PROPERTY_NAME'
                                  iv_value = 'New Value' ).
ENDIF.
```

When an entity is locked its properties are reread to make the latest data available.

3.2 Standard Interface to Access Properties of Business Objects

To access properties of BOL entities and query services (among others) use their IF_BOL_BO_PROPERTY_ACCESS interface methods.



The interface is implemented by entities, simple and dynamic query services, and parameters of dynamic queries. The interface offers a standard means to work with BOL objects.

Note

It is possible to cast many BOL objects to an instance of IF_BOL_BO_PROPERTY_ACCESS.

Syntax

```
* Get search criterion of BOL query
DATA: lv_query TYPE REF TO cl_crm_bol_query_service,
      lv_bo TYPE REF TO if_bol_bo_property_access,
      lv_city TYPE string.
lv_query = cl_crm_bol_query_service=>get_instance(
   ‘QUERY_NAME’ ).
lv_city = lv_query
   ->if_bol_bo_property_access~get_property_as_string(
   ‘City’ ).
* Short form using alias
lv_city = lv_query->get_property_as_string( ‘City’ ).
* up cast and generic access
lv_bo ?= lv_query.
lv_city = lv_bo->get_property_as_string( ‘City’ ).
* Set property of BOL entity
DATA: lv_entity TYPE REF TO cl_crm_bol_entity
lv_entity->if_bol_bo_property_access~set_property(
   iv_attr_name = ‘CITY’
   iv_value = ‘Walldorf’ ).
* Short form using alias
lv_entity->set_property( iv_attr_name = ‘CITY’
                         iv_value = ‘Walldorf’ ).
```

```
* up cast and generic access
lv_bo ?= lv_entity.
lv_bo->set_property( iv_attr_name = ‘CITY’
                     iv_value = ‘Walldorf’ ).
```

As of release CRMUIF 5.1 the interface offers methods to receive the text for a key-code value in addition to generic getter and setter methods to read and modify business objects properties.

Syntax

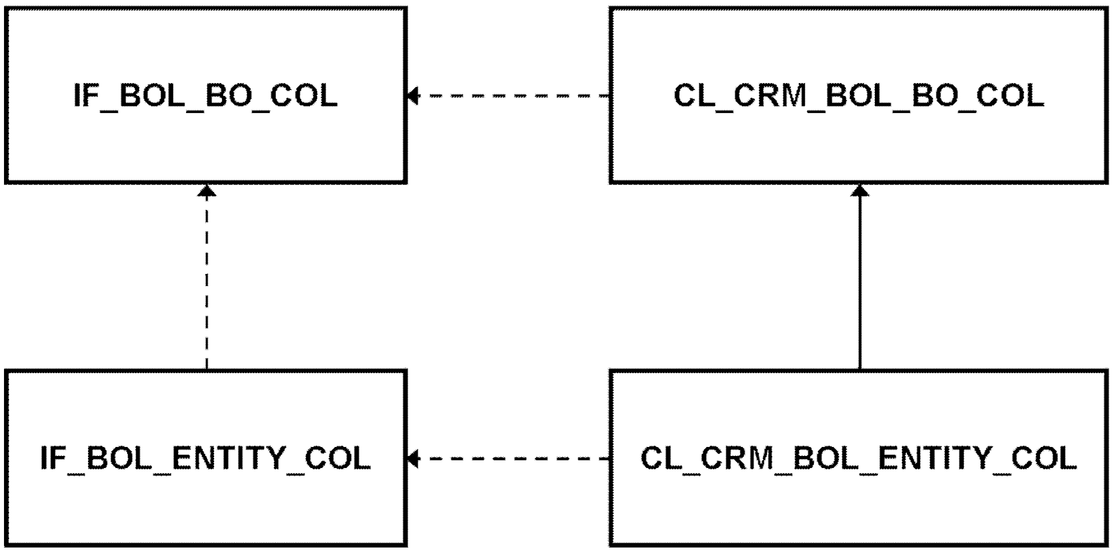
```
* Get key code
DATA: lv_person TYPE REF TO cl_crm_bol_entity,
      lv_bo     TYPE REF TO if_bol_bo_property_access,
      lv_sex    TYPE REF TO bu_sexid. “Defines values 1 =
                                       Female, 2 = Male
lv_bo ?= lv_person.
lv_sex = lv_bo->get_property( ‘SEX’ ). “Key code
* Get property text for key code:
* -> ‘Male’ or ‘Female’ translated in current language
DATA: lv_sex_text TYPE string.
lv_sex_text = lv_bo->get_property_text( ‘SEX’ ).
```

The text is taken from the domain values if available or has to be provided by the underlying GenIL component.

3.3 Working with Collections

The BOL API offers several collections for application use. The most generic collection is CL_CRM_BOL_COL and it can be used to hold all instances implementing the standard property access interface IF_BOL_BO_PROPERTY_ACCESS. The collection implements the standard collection interface IF_BOL_BO_COL. The collection CL_CRM_BOL_ENTITY_COL is especially designed for handling entities implementing the IF_BOL_ENTITY_COL collection interface for entities.

There exist several further specializations of these classes. For more information, see *Specialized Collections* [page 34].



The collection interfaces offer a bunch of methods to work with collections – e.g. GET_FIRST, GET_NEXT, GET_CURRENT, SORT, CLEAR, or ADD.

As of release SAP_ABA 7.0 collections implement the additional interface IF_BOL_BO_COL_MULTI_SEL for multi selection support. For more information, see *Single and Multi Select Support* [page 31].

All collections provide events to track the collection state. The following events are available:

- FOCUS_CHANGED: Raised, if the focus element in single selection mode changes.
- BO_ADDED: Raised, if a new object is added to the collection.
- BO_DELETED: Raised, if an object is removed from the collection.

The events also provide additional information about the object which is subject of the event if possible.

Additionally, all collections provide an auto cleanup mode where deleted entities will be automatically removed from the collection. For more information, see *Auto Cleanup Mode* [page 33].

Furthermore, collections provide a reset survival mode where the collection tries to restore all entities in the collection after a BOL reset. For more information, see *BOL Reset Survival Mode* [page 33].

3.3.1 Local and Global Iterations

All collection types support global and local iteration. In the default single selection mode, each non-empty collection has a well-defined focus object. Initially, the first object has the focus. Any global iteration moves the focus, which is published by the event FOCUS_CHANGED of the collection.

If you want to iterate on the collection without moving the focus (and without triggering time-consuming follow-up processes) you have to use local iteration. To do so, request an iterator object from the collection and use it to iterate.

Syntax

```
* Create collection
DATA: lv_collection TYPE REF TO cl_crm_bol_bo_collection,
    lv_property_access TYPE REF TO if_bol_bo_property_access,
   lv_query TYPE REF TO cl_crm_bol_query_service.
CREATE OBJECT lv_collection.
...
* Add item and make it current
lv_collection->if_bol_bo_col~insert( iv_bo    = lv_query
                                 iv_index     = 1
                                 iv_set_focus = ABAP_BOOL ).
* Global iteration
lv_property_access = lv_collection->get_next( ).     “Global
                                        “ iteration moves focus
* Local iteration
DATA: lv_iterator TYPE REF TO if_bol_bo_col_iterator.
lv_iterator = lv_collection->get_iterator( ).
lv_property_access = lv_iterator->get_first( )
WHILE lv_property_access is bound.
   lv_property_access = lv_iterator->get_next().      “Local
ENDWHILE.                      “ iteration does not move focus
```

There are two iterator interfaces. The standard interface IF_BOL_BO_COL_ITERATOR for general access is provided by all collections. Collections of CL_CRM_BOL_ENTITY_COL type also support the IF_BOL_ENTITY_COL_ITERATOR interface for more strict typed access.

Additionally to local iteration to avoid changes of the collection focus, iterators also provide features like searching by property (see *Searching* [page 27]) or filtering (see *Filtering* [page 29]).

Iterators are unmanaged sub objects of a collection. Once you retrieve an iterator the collection will not remember it. It is also not possible to determine afterwards on which collection an iterator will operate. Without this option to access the creating collection directly, iterator method GET_COPY allows to get a further iterator on the collection without knowing it.

3.3.2 Searching

Collections provide the `FIND` method to search for a given business object in the collection. If found it is returned and the focus is set to this collection entry.

The method takes two parameters, but only one is used for the search at a time. You can search by index or business object instance. The parameters are taken in the given sequence. This means that when you provide an index and an instance only the index is used.

The local iterator interfaces support the search for a single property. This option is provided by the FIND_BY_PROPERTY method. The first object whose property has the given value is returned. Neither the global nor the local collection focus is influenced by this operation.

Syntax

```
* Find by property
DATA: lv_persons  TYPE REF TO cl_crm_bol_entity_collection,
      lv_male     TYPE REF TO cl_crm_bol_entity,
      lv_iterator TYPE REF TO if_bol_bo_col_iterator.
lv_iterator = lv_persons->get_iterator( ).
lv_male = lv_iterator->find_by_property( iv_attr_name =
                                                   'SEX'
                                         iv_value = 2 ).
* Set collection focus on the entity found
lv_persons->find( iv_bo = lv_male ).
```

3.3.3 Sorting

You can sort collections using the SORT method. With this method, the collection itself is sorted and will stay in the resulting sort order. You cannot undo the sort operation.

The mandatory parameter IV_ATTR_NAME specifies the property the collection is to be sorted by. If you do not specify IV_SORT_ORDER the sort order is ascending.

Syntax

```
* Sorting by gender
DATA: lv_persons TYPE REF TO cl_crm_bol_entity_collection.
lv_persons->sort( iv_attr_name = 'SEX'
   iv_sort_order = IF_BOL_BO_COL=>SORT_DESCENDING ).
```

Note

The sorting itself is alphanumerical based on the internal format of the attribute. This means that conversion exits are not considered.

Note

Text-type components are sorted according to the locale of the current text environment.

The optional parameter IV_STABLE = ABAP_TRUE is used to perform stable sorting, which means that the relative sequence of lines that does not change in the sort key remains unchanged in the sort. Without this addition, the sequence is not retained and multiple sorting of a table using the same sort key results in a different sequence each time the table is sorted.

The optional parameter IV_USE_TEXT_RELATION = ABAP_TRUE allows you to sort using the text related to a key when the attribute is just a key and the key-text relation is known. The text used in the sorting can be retrieved via GET_PROPERTY_TEXT. For more information, see *Standard Interface to Access Properties of Business Objects* [page 23].

The following code sample shows the sorting of a collection of persons by gender. Whereas the gender key SEX (1 or 2) is used in the above example the gender text ('Male' or 'Female') is used for sorting here.

Syntax

```
* Sort collection containing UIFFlight entities
lv_flights->sort( iv_attr_name         = 'SEX'
                  iv_use_text_relation = ABAP_TRUE ).
```

The optional parameter `IV_PATH_2_SUBOBJECT` is used to sort the collection by an attribute of a related child object. The given relation or relations should be 1:1 relations. In case of a 1:N relation the first object is taken.

If more than one relation should be followed the relations need to be separated by a slash '/'. The attribute name given with `IV_ATTR_NAME` must fit in with the addressed target object. The following code sample shows the sorting of a collection of flights by the name of the customers.

Syntax

```
* Sort collection containing UIFFlight entities
lv_flights->sort( iv_attr_name = 'NAME'
   iv_path_2_subobject = 'FlightBookRel/BookCustomerRel' ).
```

When the sort table is created, relation FlightBookRel links to object UIFBooking for each flight. From there, relation BookCustomerRel links to UIFCustomer. The table is sorted by NAME which is taken from the UIFCustomer object. Since *FlightBookRel* is a 1:N relation only the first booking is considered.

You can control the sorting externally in the following cases:

- If the desired sort order is not alphanumerical
- If the desired sort order is based on not supported criteria
- If the desired sort order is based on more than one field

To do so, an instance of IF_BOL_COL_SORTING can be provided by the caller of the SORT method. During the sort process the method IS_A_GREATER_B of the instance is called whenever two values are compared. Implement this method and provide the interface to influence the sort order as needed.

Normally the actual attribute values of the given attribute `IV_ATTR_NAME` are passed as `IV_A` and `IV_B`. In case pseudo attribute name `IF_BOL_COL_SORTING=>CUSTOM` is given the whole entity instance is passed instead. This is also supported in combination with parameter `IV_PATH_2_SUBOBJECT`. So the navigation along a fixed path of relations does not need to be implemented in method IS_A_GREATER_B.

3.3.4 Filtering

Collections can be filtered with the help of an iterator. You can have any number of iterators for a collection as well as any number of filters on it at the same time. A filter is only applied via the iterator(s). Thus, the collection itself is never changed by the filtering.

The filter functionality is the same for both iterator types IF_BOL_BOL_COL_ITERATOR and IF_BOL_ENTITY_COL_ITERATOR. Therefore, all examples concentrate on the standard iterator IF_BOL_BOL_COL_ITERATOR only.

To set a filter an iterator for the collection is required (see *Local and Global Iterations* [page 26]). A filter can be set using the IF_BOL_BOL_COL_ITERATOR~FILTER_BY_PROPERTY method. You have to specify the name of the attribute and a value or value pattern as filter criterion. You can optionally choose the filter mode.

Syntax

```
* Filtering a collection of persons for females only
DATA: lv_persons TYPE REF TO cl_crm_bol_bo_collection,
      lv_filter  TYPE REF TO if_bol_bo_col_iterator.
lv_filter = lv_persons->get_iterator( ).
```

```
          lv_filter->filter_by_property( iv_attr_name = 'SEX'
                                         iv_value     = 1 ).
```

The above code sample shows the filtering of a collection of persons by gender. The gender can be "female" (attribute SEX = 1) or "male" (attribute SEX = 2). After setting the filter criterion, the iterator only operates on objects matching the given criterion, in the example persons being female. The attribute value can be given as pattern. The standard ABAP operator CP will be used for comparison.

All iterator methods respect the filter. This applies especially to the iterator methods SIZE and GET_BY_INDEX, which behave differently to their counterparts directly on the collection. You can checked if a filter is used with the public attribute `IF_BOL_BO_COL_ITERATOR~FILTER_ACTIVE`.

To set up a more complex filter you can call the FILTER_BY_PROPERTY method several times. Each call will refine the result according to the additional filter criteria.

You can delete a complex filter by deleting the criteria in the same order they were added. The DELETE_FILTER method by default only removes the last filter criterion. With the optional parameter `IV_ALL = ABAP_TRUE` you can remove all filters at the same time.

You can adapt the filter criteria of your or any given iterator using method ADAPT_FILTER. You can store filter criteria in a collection independent iterator using iterator method GET_COPY in combination with method ADAPT_FILTER.

The default filter mode is IF_BOL_BO_COL_ITERATOR=>FILTER_MODE_CACHED. In this mode, the iterator stores a filtered copy of the original collection. Therefore, operations on the filtered collection are as efficient as without filter. The downside of this approach is that the filter result might get invalid when the original collection or the contained entities are changed. Consequently, this mode should only be used in read-only mode.

The alternative filter mode is IF_BOL_BO_COL_ITERATOR=>FILTER_MODE_INTERACTIVE. Here, the iterator only stores the filter criteria and still operates on the original collection. For each iterator operation, the filter is interactively applied to the original collection computing the result. Thus, the operations get less efficient, especially operations on the whole collection, such as `GET_SIZE`. However, the result is always correct, reflecting all changes to the collection and their entities directly.

3.3.5 Changing the Collection Content

The content of a collection can be changed via the following single object operations:

• IF_BOL_BO_COL~ADD	Append an object to the collection
• IF_BOL_BO_COL~INSERT	Insert an object into the collection at a given position
• IF_BOL_BO_COL~REMOVE	Remove an object from the collection. All occurrences of the object will be removed.

You can monitor the adding and removing of objects to/from the collection via the collection events BO_ADDED and BO_DELETED. The added/removed objects are published by the events.

Additionally, the following mass operations are supported:

• IF_BOL_BO_COL~ADD_COLLECTION	Appends the content of a given collection to the collection
• IF_BOL_BO_COL~CLEAR	Removes all objects from the collection

You can also monitor the adding and removing of objects to/from the collection via the collection events BO_ADDED and BO_DELETED as well when performing mass operations. However, the added/removed objects are not published by the event in that case.

All operations might change the focus object of the collection. For more information, see *Single and Multi Select Support* [page 31].

Note

When adding an object to the collection the sort order is not automatically preserved. Filters might also get outdated. For more information, see *Filtering* [page 29].

3.3.6 Single and Multi Select Support

BOL collections support two different modes for entry selection:

- Single Selection Mode

  In this mode, it is only possible to select a single entry at a time: As soon as another entry is selected, the previous is de-selected.

- Multi Selection Mode

  In this mode, it is possible to select more than one entry at a time

You can check the current selection mode using the public collection attribute `IF_BOL_BO_COL~MULTI_SELECT`, which could be ABAP_TRUE or ABAP_FALSE, and it can be set with method IF_BOL_BO_COL~SET_MULTI_SELECT.

In single selection mode, the selected element is accessed with the following methods:

Method	**Result**
IF_BOL_BO_COL~GET_CURRENT	Returns the selected element
IF_BOL_BO_COL~GET_CURRENT_INDEX	Returns the index of the selected element
IF_BOL_BO_COL~PUBLISH_CURRENT	publishes the selected element using the IF_BOL_BO_COL~FOCUS_CHANGED event

The selected element is implicitly set with the following methods:

Method	**Result**
IF_BOL_BO_COL~FIND	Moves focus either to given index or object
IF_BOL_BO_COL~GET_NEXT	Moves focus to the next element and returns it
IF_BOL_BO_COL~GET_PREVIOUS	Moves focus to the previous element and returns it

Note

In single selection mode the current or focus element is always defined, except the collection is empty. The methods of the interface IF_BOL_BO_COL_MULTI_SEL are deactivated.

When multi selection mode is activated, you may use the methods of the interface IF_BOL_BO_COL_MULTI_SEL to mark and unmark entries and to check which entries have been marked (IF_BOL_BO_COL_MULTI_SEL~MARK, UNMARK, GET_MARKED).

In multi selection mode, the collection methods mentioned above behave differently than in single selection mode:

Method	Result
IF_BOL_BO_COL~GET_CURRENT	Returns the last selected element
IF_BOL_BO_COL~GET_CURRENT_INDEX	Returns the index of the last selected element
IF_BOL_BO_COL~PUBLISH_CURRENT	Does nothing
IF_BOL_BO_COL~FIND	Finds, eventually selects and returns an element
IF_BOL_BO_COL~GET_NEXT	Does nothing
IF_BOL_BO_COL~GET_PREVIOUS	Does nothing

Changes to the collection may change the focus. Method IF_BOL_BO_COL~ADD changes the focus when the collection was empty before. In all other cases, it can change the focus if the optional parameter IV_SET_FOCUS is given as ABAP_TRUE. The same applies to method IF_BOL_BO_COL~INSERT.

The method IF_BOL_BO_COL~CLEAR changes the focus to nothing.

Note

Special attention should be paid to the focus handling of the method IF_BOL_BO_COL~REMOVE. If the removed object has not had the focus it remains unchanged. If the focus object is removed the focus is moved to the next object in the collection (the index of the focus object remains unchanged). Only if it the last object in the collection is removed (and the collection is not empty) the focus moves to the previous object.

The following code sample demonstrates a possible coding mistake based on this behavior and the correct solution for that mistake:

Syntax

```
* Example: Remove all flights from Air Canada (AC) from the
collection

DATA: lv_flights TYPE REF TO if_bol_entity_collection,
      lv_flight TYPE REF TO cl_crm_bol_entity.

* Wrong code

lv_flight = lv_flights-> get_first( )

WHILE lv_flight is bound.

   IF lv_flight->get_property_as_string( ‘CARRID’ ) = ‘AC’.

      lv_flights->remove( iv_entity = lv_flight ).

   ENDIF.

   lv_flight = lv_flights->get_next( ).

ENDWHILE.

* Correct code

lv_flight = lv_flights-> get_first( )

WHILE lv_flight is bound.
```

```
      IF lv_flight->get_property_as_string( ‘CARRID’ ) = ‘AC’.
        lv_flights->remove( iv_entity = lv_flight ).
        lv_flight = lv_flights->get_current( ).
      ELSE.
        lv_flight = lv_flights->get_next( ).
      ENDIF.
    ENDWHILE.
```

Because of the global iteration the method REMOVE always affects the current focus element. When the focus object is removed the next object automatically gets the focus. Thus an unconditional call of GET_NEXT would leave out some objects from the check.

3.3.7 Auto Cleanup Mode

The auto cleanup mode for a collection ensures that deleted entities are automatically removed from the collection. As a consequence the (last) selected element changes automatically.

Automatic entity removal is not always necessary and this feature can lead to serious problems with memory, so by default it is inactive. To activate this function, call method IF_BOL_BO_COL~ACTIVATE_AUTOCLEANUP. The current state of the collection is indicated by the public attribute `IF_BOL_BO_COL~AUTOCLEANUP`, which is ABAP_TRUE if auto cleanup is active.

Note

If auto cleanup mode is activate, it is necessary to clear a collection (call method IF_BOL_BO_COL~CLEAR) when it is no longer needed. The garbage collector cannot delete the collection if it is referenced in the event handler table of its entities. If you do not clear unneeded collections, the affected collections remain in memory until the next BOL reset operation or session end. This can lead to serious memory issues.

3.3.8 BOL Reset Survival Mode

To free unused memory the application periodically triggers a BOL reset which deletes all BOL entities and as a consequence clears all collections with auto cleanup activated. Furthermore, the reset is propagated to the underlying GenIL components to cleanup their buffers.

Some collections need to protect their content, especially if it is essential for the further progress of the application. You can use method IF_BOL_BO_COL~SET_RESET_SURVIVAL_MODE to indicate that the root and access entities in the collection should be kept and recreated afterwards.

Note

The use of the reset survival mode limits the effect of the BOL reset and can lead to considerable runtimes of the reset. To reduce this negative impact, the recreation is delayed as of release WEBCUIF 7.0.

You can check the content of the collection for reset complaints when activating the reset survival mode. To avoid negative runtime impacts of this safety check it needs to be activated via checkpoint group BOL_ASSERTS using transaction SAAB (Assertions and Breakpoints). Recorded assert violations are collected under sub key COLLECTION_NOT_RESET_COMPLIANT.

3.3.9 Specialized Collections

Additional to the simple collection CL_CRM_BOL_ENTITY_COL that only focuses on stricter typed access to entities, there are specialized collections with additional functionalities provided.

3.3.9.1 Unique Type Collections

The specialized entity collection CL_CRM_BOL_ENTITY_COL_UT restricts the content of the collection to a single entity type. The desired type is given to the constructor of the collection and all methods adding objects to the collection respect it. Violations lead to exception CX_BOL_EXCEPTION.

Note

Method ADD_COLLECTION is not supported by this collection type.

You can convert an existing collection to a unique type collection and preserve its content and state using static method CL_CRM_BOL_ENTITY_COL_UT=>CONVERT. If the conversion is not possible it will result in exception CX_BOL_EXCEPTION.

Since the type of all entities in the collection is equal the collections offer mass operations on its content. So far only navigation is supported. Method GET_RELATED_ENTITIES returns a collection of related entities for a given relation. The collection is build out of all resulting collections of method GET_RELATED_ENTITIES on each entity in the collection.

Syntax

```
* Get all children of all entities of a collection –
standard implementation

DATA: lv_collection TYPE REF TO cl_crm_bol_entity_col,

      lv_iterator   TYPE REF TO if_bol_entity_col_iterator,

      lv_entity     TYPE_REF TO cl_crm_bol_entity,

      lv_children   TYPE Ref TO if_bol_entity_col,

      lv_result     TYPE REF TO cl_crm_bol_enity_col.

.

CREATE OBJECT lv_result.

lv_iterator = lv_collection-> get_iterator( ).

lv_entity = lv_iterator->get_first( ).

while lv_entity is bound.

   lv_children ?= lv_entity->get_related_entities(
                        iv_relation_name = ‘MyRelation’).

   lv_result->add_collection( lv_children ).

Endwhile.

* Get all children of all entities of a collection – using
unique type coll

DATA: lv_ut_collection TYPE REF TO cl_crm_bol_entity_col_ut.

lv_ut_collection = .cl_crm_bol_entity_col_ut=>convert(
lv_collection ).

lv_result = lv_ut_collection->get_related_entities(
                        iv_relation_name = ‘MyRelation’ ).
```

The collection method supports all features of the entity method. Furthermore, optional parameter `IV_MARKED_ONLY` allows restricting the set of parent entities to the marked entities in multi selection mode; in single selection mode only the focus object is taken.

3.3.9.2 Unique Instance Collections

With the specialized collections CL_CRM_BOL_UNIQUE_BO_COL and CL_CRM_BOL_UNIQUE_ENTITY_COL you can check that each BO or entity instance is only stored once in a collection. This makes it easy to collect objects in a collection without the risk of duplicates.

All methods adding objects to the collection check the uniqueness and ignore objects that are already part of the collection.

3.4 Transaction Handling

3.4.1 Transaction Context Types

Transactions are handled via transaction contexts represented by interface IF_BOL_TRANSACTION_CONTEXT. There are three different types of transaction contexts:

- Global Transaction Context

  This context covers all changes. It is retrieved via the BOL core method GET_TRANSACTION. The global transaction context is managed by the BOL core and does not change in the whole session.

- Single Business Object Transaction Context

  This context covers the changes of a single business object instance. It is easily retrieved via method GET_TRANSACTION of any entity belonging to this BO. Alternatively you can retrieve transaction contexts per root entity instance from the global transaction context via method GET_TX_CONTEXT. Single BO transaction contexts are managed by the global transaction manager. The instance associated with a root entity is unique for a transaction cycle. Once the transaction has finished a new transaction context is associated with the root entity and the prior one becomes useless.

- Custom Transaction Context

  This context is used for any scope in between the two other context types. A custom transaction context is an instance of class CL_CRM_BOL_CUSTOM_TX_CTXT, which needs to be instantiated as needed. With this, any transaction context can be added to it, including other custom transaction contexts. Custom transaction contexts are not centrally managed.

The methods of the transaction contexts always act on the whole content of a context.

BOL transactions handle all database updates synchronously.

3.4.2 Transaction Cycle

The general life cycle is the same for all three transaction context types although the creation and lifetime is different for all of them.

A transaction starts with locking the first object at the latest. The lock event is published by the BOL core and the transaction manager creates a single BO transaction context for the locked BO. This instance is stored in the transaction manager. If there has been a transaction context created before for this BO it will be used. The setting of the lock is published by this single BO transaction context via event LOCK_SET.

You can change the BO after starting the transaction. All changes made via the BOL API are collected in the BOL buffer. Calling method `MODIFY` of the BOL core communicates the changes to the underlying business logic. If changes are expected the BOL core communicates them to the transaction manager and they are saved in the associated single BO transaction context. This can be repeated many times. You can check if changes that need to be saved exist via method CHECK_SAVE_NEEDED.

Note

Pending changes in the BOL buffer are not recognized by this method.

To close the transaction cycle you can save or reset the changes. Before saving you can call the business logic for further consistence checking via method CHECK_SAVE_POSSIBLE. The actual saving of the data is initiated by calling method SAVE. This method returns a success indicator. In the case of success the transaction is closed by calling method `COMMIT`. In the unlikely case of an error during saving the transaction cycle needs to be finished with method ROLLBACK. Using method CHECK_SAVE_POSSIBLE avoids errors.

When calling `COMMIT` you can specify that you want to keep the lock via optional parameter `IV_KEEP_LOCK = ABAP_TRUE`. The lock is released in any case, but the BOL tries to lock the object again immediately.

Note

The BOL method COMMIT always forces a synchronous database update.

To reset all changes, call method REVERT. This method resets the buffer state and releases the lock. The buffer is directly synchronized with the latest saving state from the database by default. To delay the immediate synchronization, set the optional parameter `IV_SUPPRESS_BUFFER_SYNC = ABAP_TRUE`.

Note

If direct buffer synchronization is suppressed buffer objects are synchronized only at the next access. In this lazy synchronization mode, it is possible that you access an entitity that has already been deleted. This leads to an exception. To be save, you may check the entity state via method ALIVE or you program securely with a `TRY. CATCH CX_BOL_EXCEPTION. ENTRY.` construct.

After the transaction cycle a single BO transaction context becomes obsolete and cannot be used any longer. A custom transaction context can be used for another cycle, and the global transaction context starts a new cycle anyway.

3.4.3 Collective Transactions and Change Tracking

There is no monitoring of dependencies between root objects so far. It might be that a newly created root object refers to another newly generated object. If the first one is saved but the second one not, the database is in an inconsistent state. Since such dependencies are not automatically tracked, it is up to the application to handle them.

Custom transaction contexts collect single BO transactions which logically belong together. It is used to build one transaction context consisting of more than one single BO transactions.

Syntax

```
* 1. Create custom tx context

DATA: my_tx_context TYPE REF TO
         cl_crm_bol_custom_tx_context.

CREATE OBJECT my_tx_context.

* 2. Add some single BO transactions
```

```
DATA: lv_entity1 TYPE REF TO cl_crm_bol_entity,
      lv_entity2 TYPE REF TO cl_crm_bol_entity,
      lv_tx_ctxt TYPE REF TO if_bol_transaction_context.
...
lv_tx_ctxt = lv_entity1->get_transaction( ).
my_tx_context->add_tx_context( lv_tx_ctxt ).
lv_tx_ctxt = lv_entity2->get_transaction( ).
my_tx_context->add_tx_context( lv_tx_ctxt ).
* 3. Save and commit both single BO transactions together
my_tx_context->save( ).
my_tx_context->commit( ).
```

You can add other transaction contexts to your custom context using method ADD_TX_CONTEXT. This is possible for any transaction context, which means that not only single BO transactions contexts but also other custom transaction contexts may be added.

You can also monitor changes over a certain period of time and collect all associated transaction contexts. This collecting mode is activated via method START_RECORDING and deactivated via method STOP_RECORDING.

A transaction context that finishes independently is automatically removed from the custom transaction context that contains it. It is therefore not necessary to remove the transaction context explicitly using method REMOVE_TX_CONTEXT. So the custom transaction context is empty after it finishes or the global transaction finishes.

3.4.4 Tracking Transaction State

To track the state of a certain transaction context several events are offered:

- LOCK_SET: Raised only by single BO transaction context when the associated business object has been locked.
- TX_FINISHED: Raised by any transaction context when the transaction cycle has finished either via COMMIT or REVERT. In case of single BO transaction contexts, the event parameter `INSTANCE` holds the name and key of the associated root entity. As of release WEBCUIF 7.0 event parameter `LOCKS_KEPT` publishes the locking state of the BO after the end of the transaction.
- AFTER_TX_FINISHED: Raised by single BO transaction contexts after event TX_FINISHED. When performed the raising context is already removed from the global transaction manager. Thus, the event could be used for retrieving a follow-up transaction context.

3.5 Advanced Entity Features

3.5.1 Input Readiness and Entity Property Modifiers

To check if an entity attribute is read-only, you can use entity method IF_BOL_BO_PROPERTY_ACCESS~IS_PROPERTY_READONLY. It returns ABAP_TRUE if the property is not changeable and not mandatory.

The property modifier is defined for each entity property and determines the input readiness. It takes one of the following public constants defined in the interface IF_GENIL_OBJ_ATTR_PROPERTIES:

- `READ_ONLY`
- `CHANGEABLE`
- `NOT DEFINED`
- `HIDDEN`
- `MANDATORY`
- `TECHNICAL`

To access the property modifier, use entity method GET_PROPERTY_MODIFIER.

By default, it is possible to change attributes of query services.

3.5.2 Excluding Entities from the Central Modify

All new, changed, or deleted dependent objects are sent to the underlying APIs with one central call of BOL core method MODIFY. Depending on the application, this call may appear automatically at certain sync points. If entities are not ready to be sent, sending them can cause errors. In this case, you can exclude such entities from being sent.

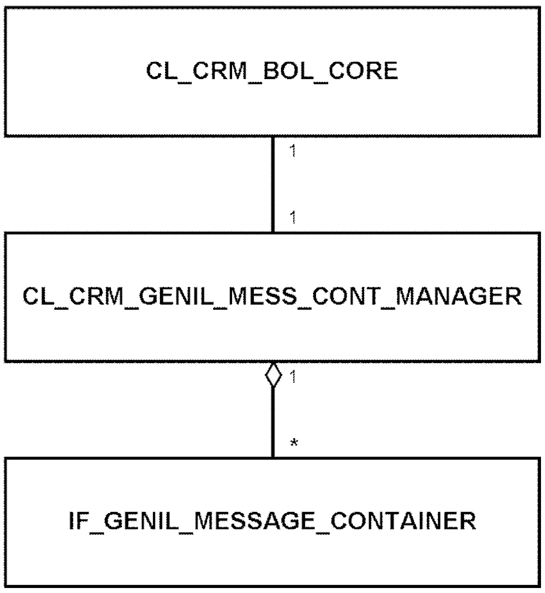
Entities are excluded in the following cases:

- A dependent entity was created but its properties were not set
- An entity was sent with MODIFY, but the changes have not been accepted by the underlying API and the entity has not been changed since. For more information, see *Business Error Handling* [page 38].
- An entity was explicitly deactivated for sending

The current status of an entity can be checked with method IS_SEND_ACTIVE on the entity instance. You can make an explicit change to the status with methods ACTIVATE_SENDING and DEACTIVATE_SENDING. If you activate or deactivate sending, this can also be propagated to aggregated child objects recursively via setting optional parameter `IV_PROPAGATE_2_DEPENDENT`. The default value is `NO_PROPAGATION`. You can change it to value `PROPAGATE_2_ALL` or `PROPAGATE_2_INACTIVE_ONLY`. Value `PROPAGATE_2_INACTIVE_ONLY` causes the recursion to stop at child objects where sending is already activated. Thus, deactivated sub-children might be preserved.

3.6 Business Error Handling

The BOL offers a message protocol to support communication of information messages, warning messages, and error messages. Messages are collected in message containers, which are handled by the message container manager: There is one message container for each access object instance and a global one for all general messages without object relation.



The message container manager can be reached using the BOL core. The following coding example shows how to access a particular message container:

Syntax

```
* Access messages of a business object
* Get message container manager
DATA: lv_mcm TYPE REF TO cl_crm_genil_mess_cont_manager.
lv_mcm = lv_bol_core->get_message_cont_manager( ).
* ... to obtain the global message container
DATA: lv_mc TYPE REF TO if_genil_message_container.
lv_mc = lv_mcm->get_global_message_cont( ).
* ... to obtain the object related message container
DATA: lv_object_name TYPE crmt_ext_obj_name,
      lv_object_id TYPE crmt_gnil_object_id.
lv_object_name = lv_order->get_name( ).
lv_object_id = lv_order->get_key( ).
DATA: lv_mc TYPE REF TO if_genil_message_container.
lv_mc = lv_mcm->get_message_cont( iv_object_name =
lv_object_name
                              iv_object_id = lv_object_id ).
* or directly
lv_mc = lv_bol_core->get_global_message_cont( ).
lv_mc = lv_order->get_message_container( ).
```

The message container interface IF_GENIL_MESSAGE_CONTAINER provides the method GET_MESSAGES for access. With this method, it is possible to filter for errors, warnings and information because it takes the message type as parameter. Possible values for the message types are defined as constants MT_ALL, MT_ERROR, MT_WARNING, MT_INFO, and MT_SUCCESS in the message container interface.

Method GET_NUMBER_OF_MESSAGES returns the number of messages of the specified type. This may be used to notify the user that messages exist.

Syntax

```
* Access messages
DATA: lv_number_of_errors TYPE int4,
      lt_error_messages TYPE crmt_genil_message_tab.
lv_number_of_errors = lv_mc->get_number_of_messages( lv_mc->mt_error ).
IF lv_number_of_errors <> 0.
  lt_error_messages = lv_mc->get_messages( lv_mc->mt_error ).
ENDIF.
```

If the underlying business logic rejects changes in case of errors, the BOL automatically deactivates sending to prevent sending the errors repeatedly. The affected entities remain in buffer state modified, valid, and sending deactivated. Thus, the user input, even if wrong, is preserved.

In this situation, the application presents the error messages to the user and allows the correction of the input. A further change of attributes will automatically reactivate sending for the entity.

3.7 Mass Operations Using the BOL Core

Many operations of the entity API can also be called via the BOL core instance. The difference in many cases is the possible amount of objects to execute the operation on. Logically, entity methods operate on the entity instance they are called on. In contrast, the corresponding BOL core method generally takes a collection of entities as input.

The following operations are supported as mass operations on a collection of entities:

- DELETE_ROOT_ENTITIES: Deletes a given set of root entities. For more information, see *Deleting Entities* [page 20].
- EXECUTE_ENTITY_METHOD: Executes an entity method on a given set of entities. For more information, see *Execution of Entity Methods* [page 21].
- EXECUTE_ENTITY_METHOD2: Executes an entity method type 2 on a given set of entities. For more information, see *Execution of Entity Methods* [page 21].

The BOL core supports these multi-object-enabled operations as well as all major operations provided by the entities directly, which makes it possible to program mainly against the BOL core. However, using the entities is more convenient.

3.8 Buffering Issues

The BOL operates on its own entity buffer, and so each of the following is buffered:

- Every entity found by a query
- Every entity read via navigation following a relation
- Each entity modification as well as the creation or deletion of dependent entities

The underlying buffers of the GenIL components (for the various business object types) are synchronized with BOL modifications when method CL_CRM_BOL_CORE->MODIFY() is called. In the CRM UIF application, this happens automatically with each roundtrip. The BOL buffer is also informed automatically about data changes in the GenIL component.

In special situations, an explicit synchronization between BOL buffer and GenIL component is necessary. These situations are described in the sections below.

3.8.1 Entity Properties

The entity method CL_CRM_BOL_ENTITY->REREAD can be used to synchronize the buffer state of the properties and property modifiers of a single entity. Since this method calls the underlying API directly, it should be used very carefully to avoid performance problems.

If the entity is new, the method does nothing. If the entity has been modified, the properties are not changed to preserve the modifications.

Note

As a result of the synchronization the BOL may detect the deletion of an entity which will result in exception CX_BOL_EXCEPTION.

The BOL call method `REREAD` instead of method GET_RELATED_ENTITIES for buffer update efficiency reasons. This only happens if the parent relation of the entity is an invalid 1:N relation and there are less than 20 objects in the current relation.

3.8.2 Entity Relations

The relation between entities is also subject of buffering. The relations may become invalid and get synchronized automatically on the next access like buffered properties. Even empty relations are buffered.

You can distinguish between cacheable relations and those that are not subject to buffering. Those are a property of the underlying object model.

For the cacheable relations, you can specify a mode for navigation. Therefore the navigation methods GET_RELATED_ENTITY and GET_RELATED_ENTITIES take an optional parameter `IV_MODE`, which may be set to one of the following constants (defined in class CL_CRM_BOL_ENTITY):

- `NORMAL`

  This is the default takes the relation from the buffer and reads it from the underlying API, if the buffer is invalid.

- `BUFFER_ONLY`

  This takes the relation only from the buffer.

- `BYPASSING_BUFFER`

  This reads the relation from the underlying API ignoring the buffer.

Non-cacheable relations are read from the underlying API, ignoring the given mode. Being non-cacheable is an unchangeable model property of a relation (IF_GENIL_OBJ_MODEL~RELATION_IS_CACHEABLE).

Note

For optimal system performance, use the default mode NORMAL. Synchronization issues that can be solved with bypassing the buffer lead to an error. These errors should be investigated. Sometimes the underlying business logic does not allow buffering. If that is the case buffering should be incorporated in the model. However, decide this measure very carefully because it is a global setting.

3.8.3 Preloading BOL Views

If you need to synchronize a larger set of entities with relations, the above methods are ineffective. For that reason, you can load a well-defined part of the object model with a given start entity or set of entities at once. This model part has to be defined as a BOL view in Customizing for *Customer Relationship Management* under *CRM Cross-Application Components → Generic Interaction Layer/Object Layer → Define Views*.

Each BOL view is assigned to a specific component set and has a unique view root, which is either a BOL root object or access object. The related object types and their relations are specified in a table.

After completing the customizing, method PREFETCH_VIEW can be called on the BOL core. It takes the view name and a collection of view root entities as input. The model part defined by the view is read for each entity in the collection and stored in the buffer effectively.

You can also apply a view directly to a search result of a query service. Set the name of the view on the query service instance with method SET_VIEW_NAME.

3.8.4 Locking with Synchronization

After setting a lock it is sometimes necessary to synchronize the BOL buffer for the locked entities with the current database state for the following reasons:

- The entity could have been changed by someone else since the last read operation.
- The last lock attempt failed, because the entity was already locked. Due to the failure, the entity was set to read-only mode. When trying to lock the entity again, the property modifiers have to be refreshed.

Method CL_CRM_BOL_ENTITY->LOCK takes an optional parameter `IV_REREAD`. By default, it is set to `ABAP_FALSE`. If it is set to `ABAP_TRUE` the full aggregation hierarchy of the locked root entity is invalidated in the buffer and synchronized at the next access.

Note

With display mode support, this synchronization always takes place. For more information, see *Display Mode Support* [page 23].

3.8.5 Buffering of Query Results

Queries can return a big amount of objects and therefore can considerably increase the memory consumption of the BOL buffer. The BOL reset is the only way to clear the BOL buffer, but it is not easy to control the memory consumption when frequent searches happen. However, many query services return Query Result Object as aggregation of actual entities for an efficient result display.

As of release WEBCUIF this can be further leveraged by not adding them to the central BOL buffer. This now is the default setting for simple and dynamic query services. Adding the result to the central buffer has to be enforced by setting optional parameter `IV_ADD_QRESOBJ_2_BOL_BUFFER` of method GET_QUERY_RESULT to ABAP_TRUE.

Without registration in the central buffer, query result object instances live with result collection, thus their lifetime is controlled by the application program. This also means that the object instances are no longer unique.

3.9 BOL Reset

Since the BOL buffer and the message services permanently collect but not release information the amount of data constantly grows with time. Therefore a mechanism is necessary to get rid of all the buffered/collected data.

Currently, the only possibility is the BOL core reset using method CL_CRM_BOL_CORE->RESET. This method clears all known buffers and messages. After the method has run you can also call the ABAP garbage collector explicitly to finally remove all freed objects.

In the CRM UIF application, a BOL reset is automatically triggered by the framework whenever the user switches work center by clicking in the navigation bar.

All entity instances are useless after the BOL reset. Further operations on them cause runtime exceptions. This also applies to entities contained in collections for which the reset survival mode is activated (see *BOL Reset Survival Mode* [page 33]). The reset survival mode does not preserve the actual entity instance. Only the related object is memorized and a new entity instance is created for it. To keep the current entity instances, you can provide an exception list when calling method `RESET`.

The BOL core raises event RESET_TRIGGERED when the BOL reset starts and event RESET_FINISHED when it finishes. During the BOL reset most BOL API operations are not possible. The state of the BOL core instance can be checked via its public attribute `RESET_RUNNING`.

The BOL reset clears all entities from the buffer. For these entities, event DELETED is raised.

> Note
>
> The event DELETED being raised during BOL reset does not mean that the object is actually deleted.

The BOL reset also triggers the reset of loaded GenIL components, so it might be necessary to perform it to close connections to remote systems when closing a session. Restoring data after the reset can be prevented by setting optional parameter `IV_NO_RESTORE` to `ABAP_TRUE`.

4 Interface Classes

4.1 Core

CL_CRM_BOL_CORE is the most important class of the BOL Application Programming Interface (API). There is only one instance of it within each session and it cannot be lost or deleted. The BOL core is the central service provider for API classes and it communicates with the underlying business object implementation. You can use the core services directly, but it is strongly recommended to use the more comfortable API classes, such as query services or entities.

4.2 Query Services

Generic query services are provided by the classes CL_CRM_BOL_QUERY_SERVICE and CL_CRM_BOL_DQUERY_SERVICE. Each instance corresponds to a distinct query object. It stores query parameters but does not aggregate the query result.

4.3 Entities

The class CL_CRM_BOL_ENTITY is a generic representation for business object segments in the BOL. Each instance is a unique representation for a specific part of a business object. You can access data using interface IF_BOL_BO_PROPERTY_ACCESS. Entities are centrally managed and cached.

4.4 Collection

A collection or list of generic objects is represented by interfaces IF_BOL_BO_COL and IF_BOL_ENTITY_COL and underlying classes CL_CRM_BOL_BO_COL and CL_CRM_BOL_ENTITY_COL (among others). A collection provides global iteration, which changes the global focus. Using the local iterators does not change the global focus. It is possible to add and remove objects that are referenced, but not owned by the collection.

4.5 Transaction Context

The transaction context is represented by interface IF_BOL_TRANSACTION_CONTEXT. Depending on the actual instance of the interface, the scope of the represented transaction is a single root object instance, all modified root object instances, or any explicitly built subset in between.

The BOL core aggregates the global transaction context, which includes all modified root objects. The global transaction context logs all modifications automatically. A single BO transaction context can be accessed using the entity it belongs to.

5 Checkpoint Groups

To highlight important places for debugging and to introduce expensive consistency checks, some checkpoint groups have been introduced. Checkpoint groups are activated with transaction SAAB (Assertions and Breakpoints) in the development environment.

5.1 BOL Checkpoint Groups

The following BOL checkpoint groups are available:

- BOL_ASSERTS

  This checkpoint group includes expensive checks for inner consistency and for the correct use of the display mode. If the checkpoint is activated, any attempt to change entities in display mode is detected. You can activate this checkpoint group in development and test systems to get early notice of problems related to the BOL usage.

- BOL_MODIFY_WATCH

  This checkpoint group defines important breakpoints, which can be used to track the data transport from the business object to the generic interaction layer (GenIL), the merge back of data returned into the BOL buffer, ID adjustments for new entities, and buffer invalidation of changed entities.

- BOL_COLL_AUTOCLEAN

  This checkpoint group activates the auto cleanup mode for collections as default. This may lead to memory problems, so it should only be used for testing purposes or compatibility purposes with framework versions SAP_ABA 7.0 and older.

5.2 Related Checkpoint Groups

For detailed investigations and debugging, it is required to break in the GenIL, which gives access to data persistency.

The following related checkpoint groups are available:

- GENIL_DC_CHECKS

  This checkpoint group activates consistency checks for the data containers.

- GENIL_GENERAL_CHECKS

  This checkpoint group activates general checks in the GenIL layer.

- GENIL_MODEL_CHECKS

  This checkpoint group activates consistency checks for the object models of the used GenIL components.

- GENIL_LOCK

  This checkpoint group stops the ABAP debugger if an entity is to be locked.

- GENIL_SAVE

  This checkpoint group allows investigations of the save process.

- GENIL_READ

  This checkpoint group breaks in the read method of the GenIL core CL_CRM_GENERIC_IL_NEW. It allows detailed investigations of the communication with the underlying GenIL component responsible for a specific business object.

- GENIL_MODIFY

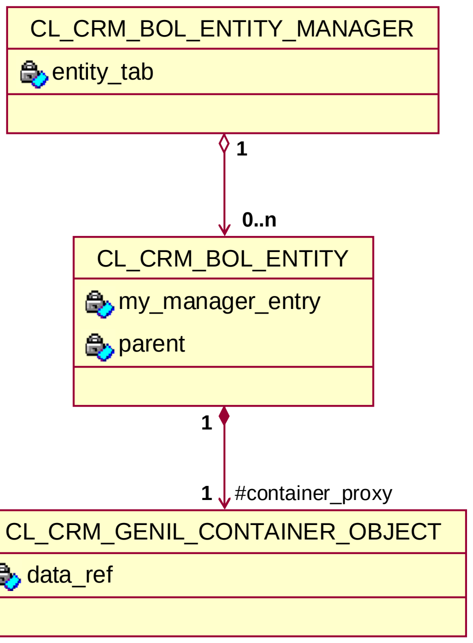
  This checkpoint group breaks in the modify method of the GenIL core.

6 Inner Details and Debugging

This section presents further aspects of the business object layer (BOL). The UML diagrams below show the connection between the classes involved and their main attributes.

6.1 BOL Entities

BOL entities are managed by the BOL entity manager and use classes of the GenIL to hold their data.

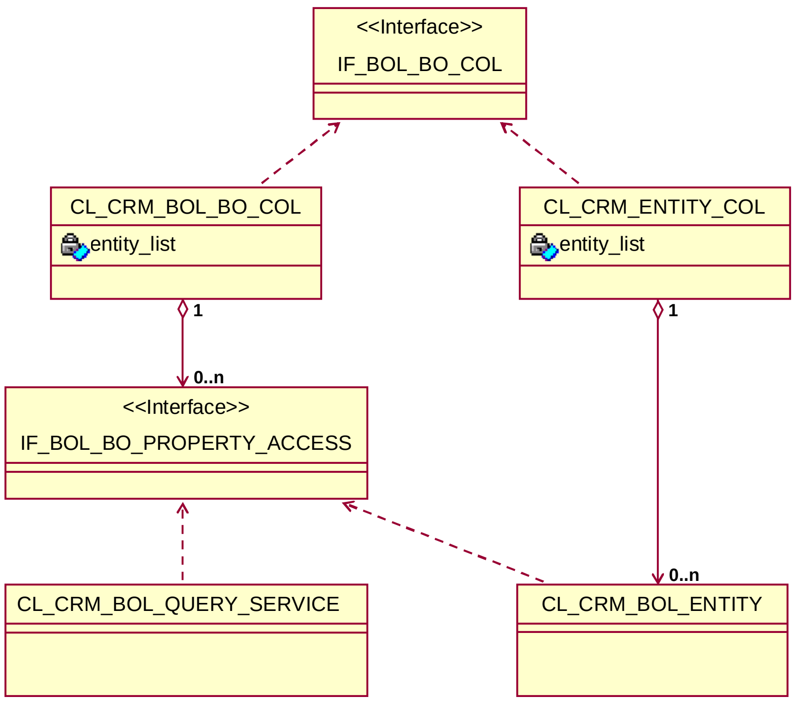


In the diagram above:

- All entities are managed by the entity manager: It assures the uniqueness of entities and buffer access.
- Each entity holds a reference to its manager entry. The manager entry includes values such as the invalid and delta flags. Holding these values in the ENTITY_TAB of the entity manager enables efficient BOL table operations on all entities, e.g. CL_CRM_BOL_CORE->MODIFY.
- An entity is a wrapper for a GenIL container object CL_CRM_GENIL_CONTAINER_OBJECT belonging to a data container. This object is referred to as CONTAINER_PROXY and holds the attributes, properties, and relations of an entity.

6.2 Collections

Various collections reference a set of BOL entities and offer convenient access to their properties.



In the diagram above:

- A collection is represented by interface `IF_BOL_BO_COL` and can either be an entity collection CL_CRM_ENTITY_COL or the generalized business object collection CL_CRM_BOL_BO_COL.
- A BO collection can hold entities (CL_CRM_BOL_ENTITY) and query services (CL_CRM_BOL_QUERY_SERVICE).
- Access to the properties of BOL objects is offered by the interface IF_BOL_BO_PROPERTY_ACCESS with its standard access methods, such as IF_BOL_BO_PROPERTY_ACCESS~GET_PROPERTY, GET_PROPERTY_AS_STRING or SET_PROPERTY.