SAIL Programming Guide

Applies to:

Social Media ABAP Integration Library
SAP NetWeaver 7.02 SP11 or greater
SAP NetWeaver 7.30 SP07 or greater
SAP NetWeaver 7.31 SP03 or greater

Summary

This document gives you a first insight into the programming with the SAIL API to integrate a collaboration platform into your application. The second part of the document throws a light on how to tailor the SAIL API to match more sophisticated requirements.

Authors: Mathias Roeher
Regina Weidemann
Company: SAP AG
Created on: 16 October 2012

Author Bio

Mathias Röher has been working at SAP AG for donkey's years and loves programming.
At the time of publication, he is a member of the team that created the SAIL API.

Regina Weidemann is a transplanted California girl. She has been a translation specialist at SAP AG since 2007.

Table of Contents

Introduction

Motivation

The Social Media ABAP Integration Library (SAIL) is part of SAP NetWeaver and the first version was delivered with SAP_BASIS 7.31 SP03, 7.30 SP07 and 7.02 SP11.
You use this library to integrate collaboration features into your application. The first part of this document guides you through the first steps of interaction with SAIL. The second part shows some more advanced options for tailoring the SAIL API to your needs.

Terms

Platform type	Identifier for a collaboration platform
Platform	Server specific to a collaboration platform
SAIL application	The application is used to tie a set of Customizing entries to a platform type. This set of Customizing entries binds the backend application (for example, CRM) and its specific implementation to the collaboration platform (by using a specific OAuth application registration, for example).
SAIL application context	Key defined by an application (such as CRM) to distinguish between settings within a SAIL application.
SAIL application key (abbr.: application key)	Combination of platform type, SAIL application and SAIL application context; identifies a SAIL API instance.

Overview

API Structure

The API consists of two layers: The **object layer** and the **infrastructure layer**. In most cases you will have to deal with the object layer only; in some scenarios you may add or replace functionality by modifying the methods (the fat arrows in the illustration below), and only in very rare cases will you make use of the possibility to replace parts of the infrastructure layer.
The methods serve as mediators between the object and the infrastructure layer. You can equate a method like this with the REST endpoint of the collaboration platform. It transforms your parameters into the REST call presentation and the results of the call into an ABAP representation.
The BAdI implementations serve three purposes: User mapping (backend user into collaboration platform user and back), authentication method, and item handling (more on this later).

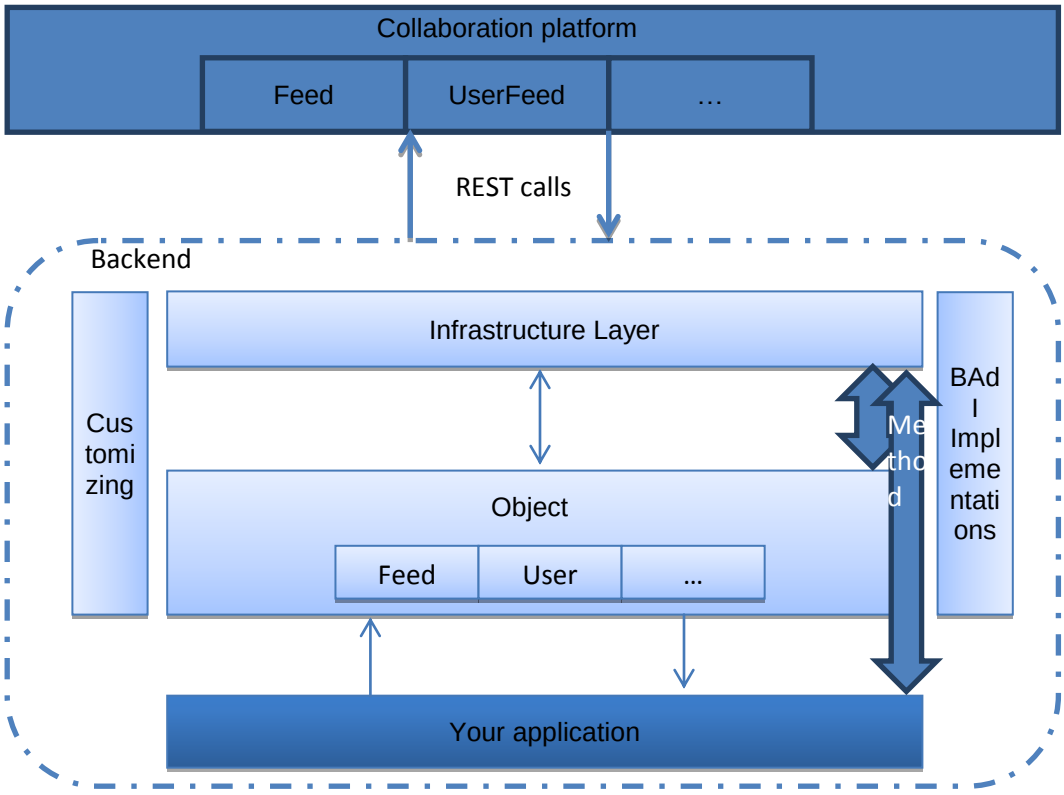


Figure 1: SAIL API Structure

SAIL Context

You access each single collaboration platform within a context:

- PLATFORM TYPE
  Names the collaboration provider you want to connect to: SAP StreamWork, Google, Twitter…
- PLATFORM
  This value identifies the server of the platform type, if applicable; for example, you can switch between the test and the productive server when you access the StreamWork provider.
- SAIL APPLICATION
  The application basically defines a complete SAIL instance for a platform type - this means the Customizing of both the object and the infrastructure layers, as well as the values needed for the connection from the backend to the collaboration provider. In other words, it binds a backend integration scenario to a concrete platform type. The default binding is provided through the application SAIL.
- SAIL APPLICATION CONTEXT
  An application context is a string value that can be used for additional fine tuning within an application.

The platform serves as a switch and is part of the Customizing or the Customizing access classes only. The other three values define the SAIL Application key (hereafter abbreviated as ***application key***).

Object Layer

The object layer provides proxies for the objects that are available on the collaboration platform: For example Feeds, Users, Activities and so on for SAP StreamWork. These proxies can be used by your coding to integrate them into your application.
A single point of entry is provided for accessing the StreamWork layer through the object layer, the interface IF_STW_API. This interface defines a single instance of a SAIL context mentioned above, identified by the SAIL application, and solely to be used for accessing SAP StreamWork; each roll area can contain exactly one SAIL API instance per SAIL application.

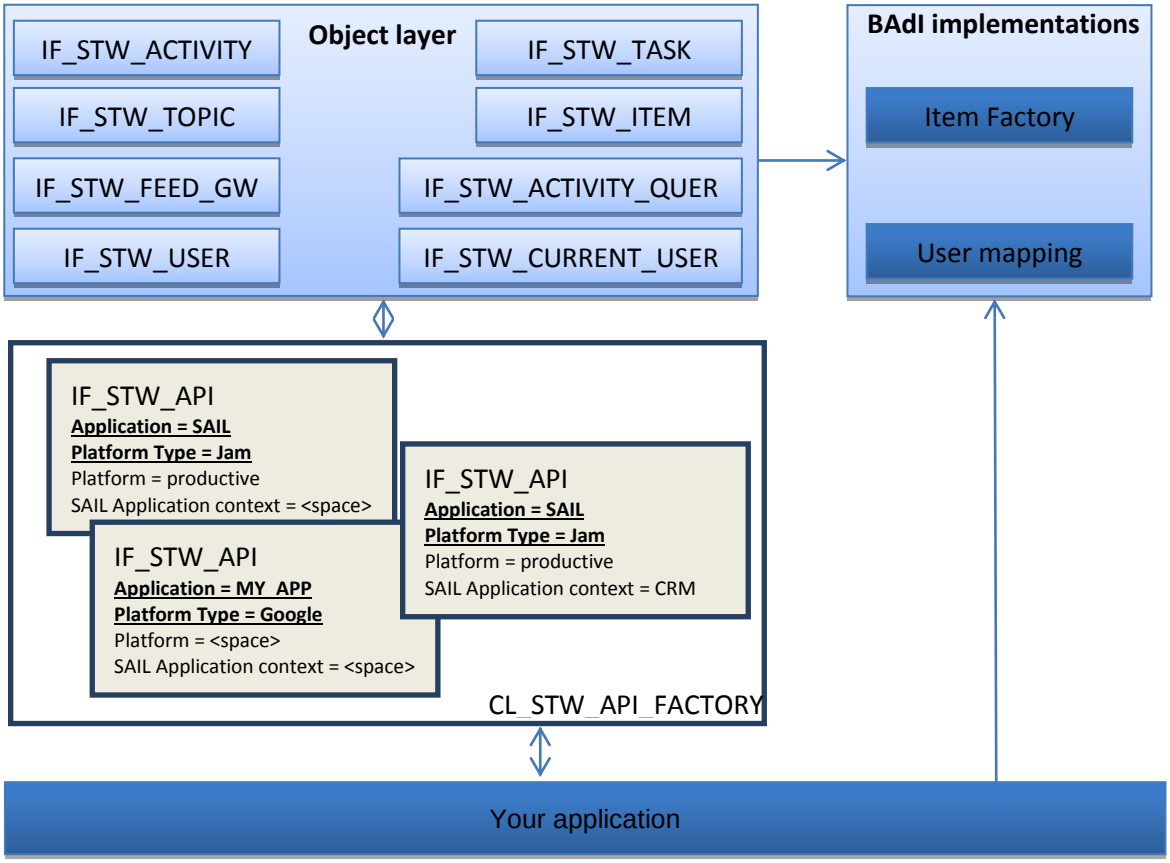


Figure 2: Object Layer

The main package of the object layer is S_STW_MAIN.
All objects have the string 'STW' in their names, as shown in the following table:

Interfaces	IF_STW_
Classes	CL_STW_
DDIC objects, message classes	STW_
Enhancement spots	STW_
Reports	RSTW_
Each other object type	STW

To see the object layer in action, you can use the reports starting with RSTW_LIBRARY_TEST*.

Infrastructure Layer

The infrastructure layer provides the frame needed to perform REST calls and it grants access to Customizing. As in the object layer, a single point of entry, IF_CLB_LIB, is provided and there can be only one instance per roll area for each combination of platform type and application.

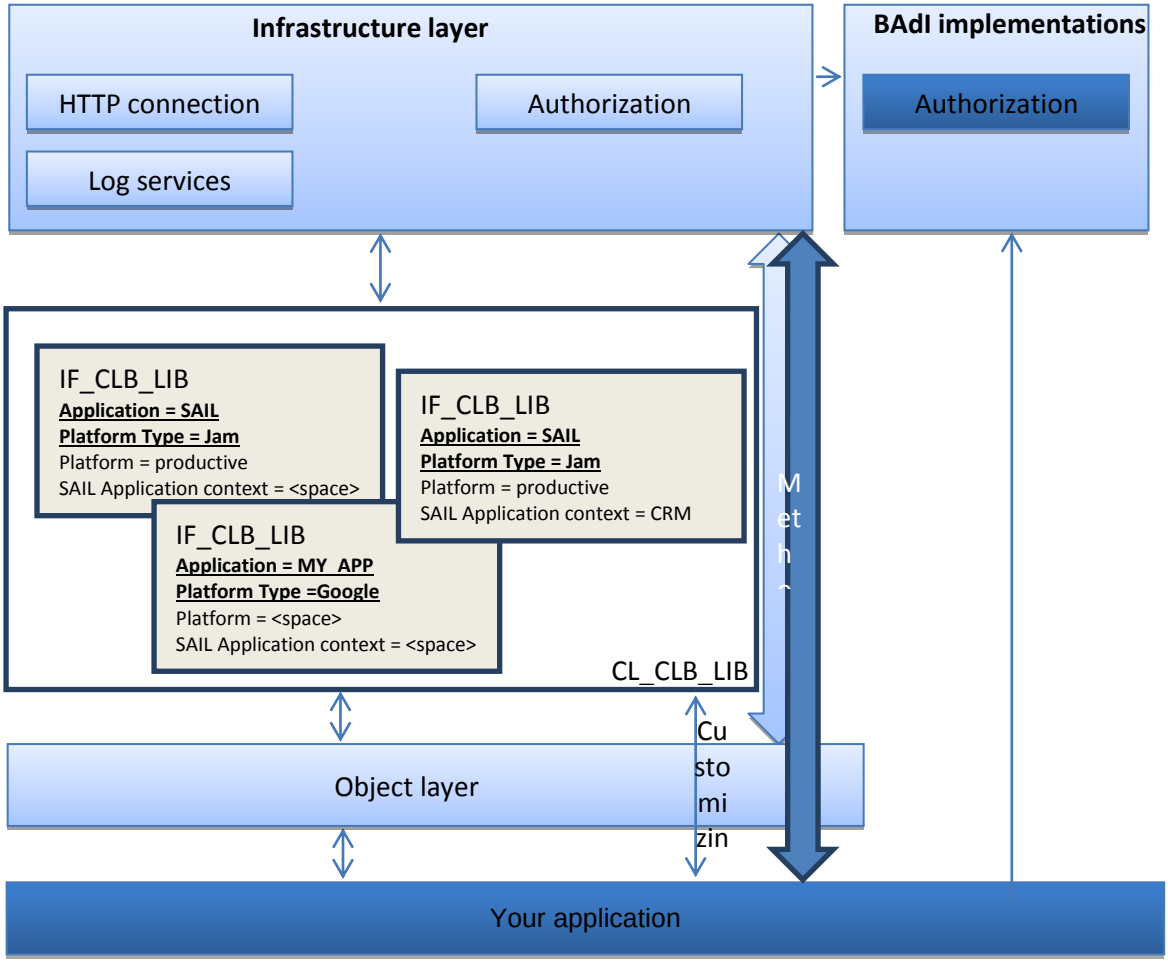


Figure 3: Infrastructure Layer

The main package of the object layer is S_CLB_MAIN.
All objects have the string 'CLB' in their names, as shown in the following table:

Interfaces	IF_CLB_
Classes	CL_ CLB _
DDIC objects, message classes	CLB _
Views	CLBV_
View cluster	CLBVC_
Enhancement spots	CLB _
Reports	R CLB _
Each other object type	CLB

Customizing Access

Customizing can be found in package S_CLB_CUST. Three objects should be emphasized:

The main object of interest for a developer is the interface IF_CLB_CUST_ACCESS because it grants access to all Customizing objects.

The second interface that needs to be mentioned is IF_CLB_CONFIGURATION. It is implemented by class CL_CLB_CONFIG and provides the convenience methods that are needed for the standard processes – this interface works within the SAIL context and therefore requires the application key.

The third interface is IF_CLB_CUST_QUERIES. It wraps multiple or complex SELECTs into single methods. The queries provided here work without the SAIL context.

```
DATA:
  lo_cust_access  TYPE REF TO if_clb_cust_access,
  lo_cust_queries TYPE REF TO if_clb_cust_queries,
  lo_cust_conf    TYPE REF TO if_clb_configuration.

  lo_cust_access = cl_clb_cust_access=>s_get_instance( ).
  lo_cust_queries = cl_clb_cust_queries()=>s_get_instance( ).
  TRY.
    lo_cust_conf = cl_clb_lib=>s_get_instance(<application key> )->get_configuration( ).
  CATCH cx_clb_process.
    …
  ENDTRY.
```

Programming with SAIL

Use the Collaboration Objects

When not stated otherwise, assume that we are working in the following context:

Platform type = StreamWork
Platform = productive
SAIL Application = SAIL
Application context = <space>

The single point of entry into the object layer is assumed to be the global variable GO_API:

```
DATA:
  go_api  TYPE REF TO if_stw_api.

  TRY.
    go_api = cl_stw_api_factory=>s_create( iv_application_id = 'SAIL' ).
    IF go_api IS NOT BOUND.
      " add your error handling
    ENDIF.
  CATCH cx_stw_exception.
    " runtime error only when the application ID is empty
  ENDTRY.
```

Activities

➔ The report RSTW_LIBRARY_TEST_ACTIVITY illustrates the options you have for this collaboration object and how to use it in your code.

An activity can be seen as a canvas for collaboration tasks; it is identified by its ID. Therefore, if you create an activity, you will get the new activity ID back. Store it somewhere, if you want to use it for later access.

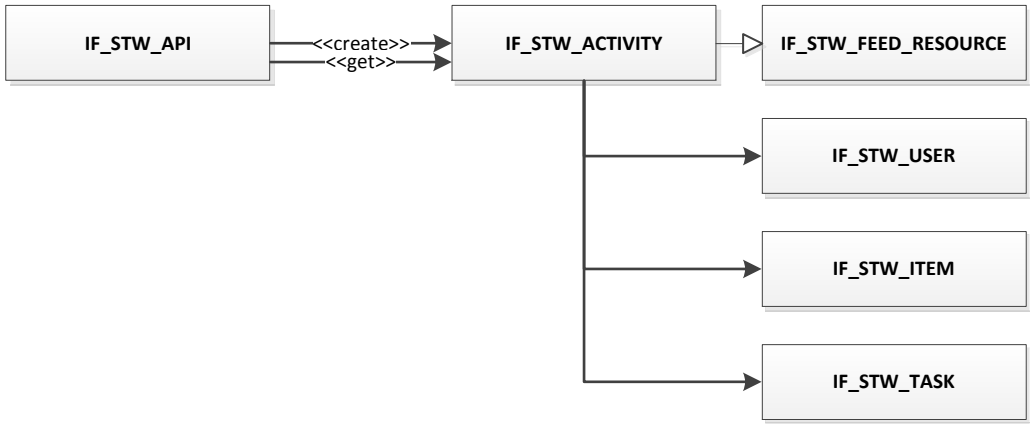


Figure 4: Main Object Relationships for Activity

If you work with an activity, the following table shows the most common tasks you may want to perform:

Task	Remarks
Create activity	Create a new activity with you as its owner. After the activity has been created, store the activity ID for later access.
Invite participants	You can invite a number of participants at the same time. The state of the invitations will be reported (already a participant, already invited, not invited due to an error).
Set owner	Change the owner of the activity.
Remove participant	Remove a participant from an activity.
Create a task	Create a new task for the activity. A task is simply an assignment of a work item to one or more users, with a task description and a due date.
Create item	Create a new item for the activity. An item is a collaborative object, for example, a table is presented as a table item, text as a text item, and so on.

A comprehensive overview can be found in the test report. The class CL_MAIN shows all the functions you can perform with the interface IF_STW_ACTIVITY.
Activities use items and tasks for collaboration purposes; for more information, please see the corresponding chapter below.

Create activity

```
DATA:
  lo_activity    TYPE REF TO if_stw_activity,
  lv_activity_id TYPE stw_activity_id.

lo_activity = go_api->create_activity( iv_name    = 'Activity name'
                                       iv_purpose = 'Short description' ).
lv_activity_id = lo_activity->get_id( ).
```

Get activity by ID / Invite participants

```
DATA:
  lo_activity          TYPE REF TO if_stw_activity,
  ls_invitation_result TYPE if_stw_activity=>ts_invitation_result.

" iv_activity_id is the ID of an activity we have created before
lo_activity = go_api->get_activity( iv_activity_id ).

" it_sw_user is a table of type if_stw_user=>tt_sw_user that
" contains the user instances for all users that will be invited
lo_activity->invite_participants(
  EXPORTING
    it_sw_user           = it_sw_user
  IMPORTING
    es_invitation_result = ls_invitation_result ).
```

Set owner

```
DATA:
  lo_activity TYPE REF TO if_stw_activity,
  lo_user     TYPE REF TO if_stw_user.

" iv_activity_id is the ID of an activity we have created before
lo_activity = go_api->get_activity( iv_activity_id ).

lo_user = go_api->get_user( iv_sw_user_id = 'Collaboration user ID' ).

lo_activity->set_owner( io_owner = lo_user ).
```

Create task

```
DATA:
  lo_activity TYPE REF TO if_stw_activity,
  lv_task_id  TYPE stw_task_id.

" iv_activity_id is the ID of an activity we have created before
lo_activity = go_api->get_activity( iv_activity_id ).

" it_participants is the list of users that will participate in the task
lo_task = lo_activity->create_task(
                  iv_title       = 'Buy keyboard'
                  iv_description = 'Look for self-correcting keyboard'
                  iv_due_date    = '20991223'
                  it_participant = lt_participants ).

" the task ID can be used for later retrieval
lv_task_id = lo_task->get_id( ).
```

Get a list of activities

```
DATA:
  lo_activity_query TYPE REF TO if_stw_activity_query,
  lt_activities     TYPE if_stw_activity=>tt_activity.

lo_activity_query = go_api->get_activity_query( ).

" it_activity_ids is a table of activity IDs
" lt_activities is a table of if_stw_activity-instances
lt_activities = lo_activity_query->query_activities_by_id( it_activity_ids ).
```

Feeds

➔ The report RSTW_LIBRARY_TEST_FEED illustrates the options you have for this collaboration object and how to use it in your code.

A feed is related to three resources: Activities, users, and topics. Feed entries result from actions that took place in the context of one of these resources and from status updates a user posts to one of the resources.

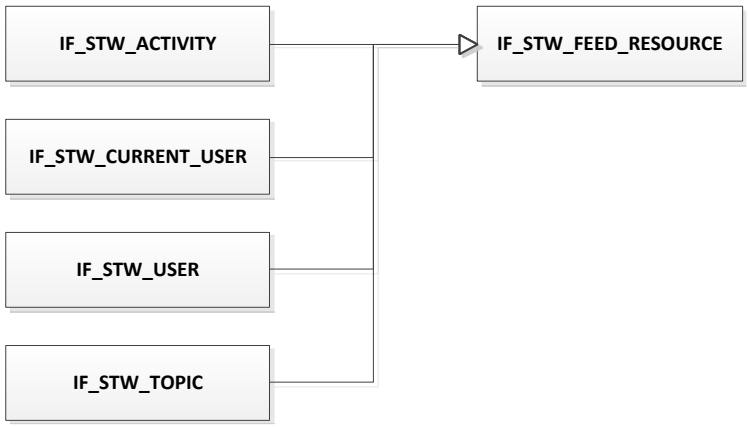


Figure 5: Main Object Relationships for Feeds

If you work with feeds, the following table shows the most common tasks you may want to perform:

Task	Remarks
Retrieve feed	You can read the feed from one of the feed resources.
Post feed entry / status update	You can explicitly post a feed entry; this can be done by posting a status update, for example, "I'm in the mood for dancing" to the current user's feed.
Follow/Unfollow	You can receive the feed of a feed resource by following this resource. If you don't want to receive it any longer, you can opt out by unfollowing.

A comprehensive overview can be found in the test report. The class CL_MAIN shows all the functions you can perform with the interface IF_STW_FEED_RESOURCE.

Retrieve feed

```
DATA:
  ls_feed        TYPE stw_s_feed,
  lo_feed_filter TYPE REF TO if_stw_feed_filter,
  lo_activity    TYPE REF TO if_stw_activity.

" iv_activity_id is the ID of an activity we have created before
" and whose feed we want to read
lo_activity = go_api->get_activity( iv_activity_id ).

" the feed filter puts some restrictions onto the result
" here we say that we want to see the important feed entries only
lo_feed_filter = go_api->create_feed_filter( ).
lo_feed_filter->set_important_only( 'X' ).


ls_feed = lo_activity->get_resource_feed( lo_feed_filter ).
```

Post status update

```
DATA:
  lo_sw_user     TYPE REF TO if_stw_current_user,
  lv_activity_id TYPE stw_activity_id.

" we are going to post an update into our personal feed
lo_sw_user = go_api->get_current_user( ).

lo_sw_user->if_stw_feed_resource~post_status_update(
               iv_status_update = 'I'm in the mood of dancing' ).
```

Follow/Unfollow

```
DATA:
  lo_activity TYPE REF TO if_stw_activity.

" iv_activity_id is the ID of an activity we have created before
lo_activity = go_api->get_activity( iv_activity_id ).

lo_activity->follow( ).
lo_activity->unfollow( ).
```

Topics

➔ The report RSTW_LIBRARY_TEST_TOPIC illustrates the options you have for this collaboration object and how to use it in your code.

Topics act as containers that are able to collect feed entries. If, for example, you associate a topic with a business object, you can post messages (see post feed entry from the feed chapter) to this topic. In this way, the topic acts as a proxy for the business object on the collaboration platform side.

Figure 6: Main Object Relationships for Topics

If you work with feeds, the following table shows the most common tasks you may want to perform:

Task	Remarks
Create topic	Create a new topic
Post feed entry / status update	Post a feed entry to the topic
Follow/Unfollow	(Un)Follow a topic
Delete topic	Delete a topic (for example when an associated business object is deleted)

A comprehensive overview can be found in the test report. The class CL_MAIN shows all the functions you can perform with the interface IF_STW_TOPIC.

Create topic

```
DATA:
  lo_topic    TYPE REF TO if_stw_topic,
  lv_topic_id TYPE stw_topic_id.

" a topic has a description and it can be linked to
" some external ID (our business object for example)
" and it be associated with a URL
lo_topic = go_api->create_topic(
                    iv_topic_description          = 'My sensational topic'
                    iv_topic_ext_ent_id           = 'External Entity ID'
                    iv_topic_ext_ent_linkback_url = 'Linkback URL' ).

lv_topic_id = lo_topic->get_id( ).
```

Post status update

```
DATA:
  lo_topic TYPE REF TO if_stw_topic.

" iv_topic_id is the ID of the topic we want to update
lo_topic = go_api->get_topic( iv_topic_id ).

lo_topic->if_stw_feed_resource~post_status_update(
               iv_status_update = 'Been there, done that' ).
```

Follow/Unfollow

```
DATA:
  lo_topic TYPE REF TO if_stw_topic.

" iv_topic_id is the ID of the topic we want to update
lo_topic = go_api->get_topic( iv_topic_id ).

lo_topic->follow( ).
lo_topic->unfollow( ).
```

Delete topic

```
DATA:
  lo_topic TYPE REF TO if_stw_topic.


" iv_topic_id is the ID of the topic we want to update
lo_topic = go_api->get_topic( iv_topic_id ).

lo_topic->delete_topic.
```

Users

➔ The report RSTW_LIBRARY_TEST_SW_USER illustrates the options you have for this collaboration object and how to use it in your code.

Nearly everything in a collaboration scenario happens in the context of a user – in most cases the current user. Within SAIL, a user has two IDs: The backend user ID and the user ID on the collaboration platform. Although the collaboration platform user ID is often an e-mail address, this is not necessarily the case and therefore SAIL distinguishes between the two.
To map the backend user ID to the collaboration platform user ID and back, a BAdI is used.
Since user maintenance is an administrative task, there are no functions for modifying user data. All functionality that can be accessed through SAIL API just queries the attributes of a user.

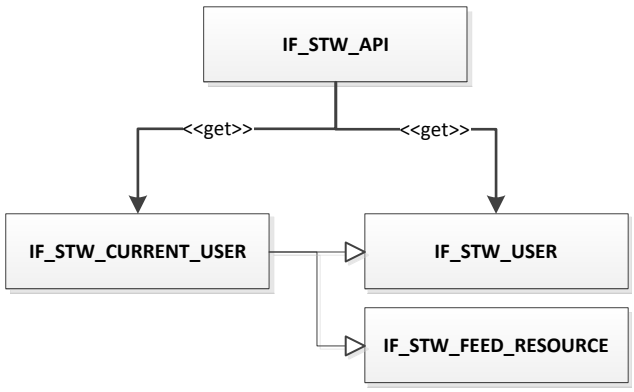


Figure 7: Main Object Relationships for Users

If you work with users, the following table shows the most common tasks you may want to perform:

Task	Remarks
Retrieve (current) user	Read the data of a user
Post feed entry / status update	Post a status to the current user's feed.
Follow/Unfollow	(Un)Follow a user
Get picture	Get a user's picture from the user's profile on the collaboration platform

A comprehensive overview can be found in the test report. The class CL_MAIN shows all the functions you can perform with the interfaces IF_STW_USER and IF_STW_CURRENT_USER.

Retrieve (current) user

```
DATA:
  lo_current_user TYPE REF TO if_stw_current_user,
  lo_user         TYPE REF TO if_stw_user.

lo_current_user = go_api->get_current_user( ).

" iv_sw_user_id is the user ID on the collaboration platform
lo_user = go_api->get_user( iv_sw_user_id = iv_sw_user_id ).

" iv_backend_user_id is the user's ID on the backend system
lo_user = go_api->get_user( iv_backend_user_id = iv_backend_user ).
```

Post status update

```
DATA:
  lo_current_user TYPE REF TO if_stw_current_user.

lo_current_user = go_api->get_current_user( ).

lo_user->if_stw_feed_resource~post_status_update(
               iv_status_update = 'Been there, done that' ).
```

Follow/Unfollow

```
DATA:
  lo_feed_resource     TYPE REF TO if_stw_feed_resource.

" iv_sw_user_id is the user ID on the collaboration platform
lo_feed_resource ?= go_api->get_user( iv_sw_user_id = iv_sw_user_id ).

lo_feed_resource->follow( ).
lo_feed_resource->unfollow( ).
```

Get picture

```
DATA:
  lo_current_user TYPE REF TO if_stw_current_user,
  lv_url          TYPE string.

lo_current_user = go_api->get_current_user( ).

" the URL can, for example, be used as source for an image control
lv_url = lo_current_user->get_sw_profile_picture_url( ).
```

Items

➔ The report RSTW_LIBRARY_TEST_ITEM illustrates the options you have for this collaboration object and how to use it in your code.

Items form different kinds of data representation and types of collaboration – from simple text items to complex tables. Therefore, as an activity is the canvas for collaborations, items always exist in the context of an activity.
Out of the box SAIL supports three types of items: a generic item, text items, and file items. A BAdI can support more specialized items.

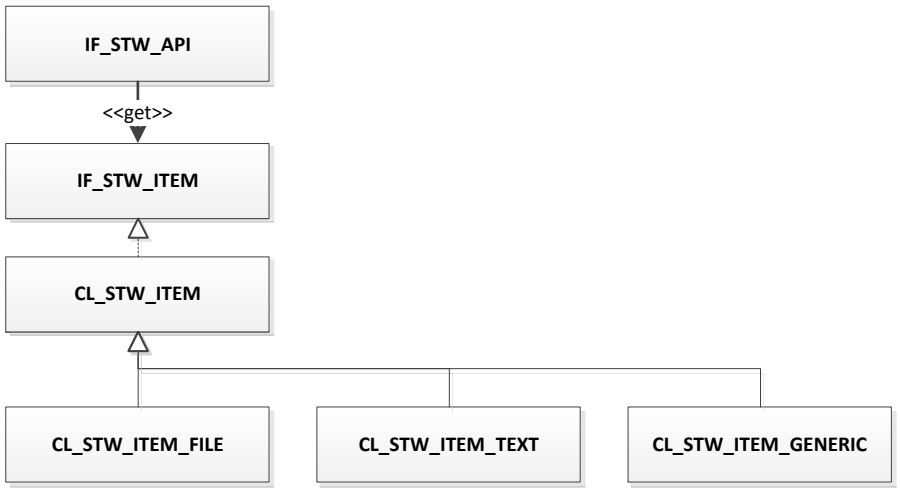


Figure 8: Main Object Relationships for Items

If you work with items, the following table shows the most common tasks you may want to perform:

Task	Remarks
Retrieve item	Retrieve a single item
Create comment	Add a comment to an item
Create item	Create a new item
Retrieve all items of an activity	Read all items that belong to an activity.

A comprehensive overview can be found in the test report. The class CL_MAIN shows all the functions you can perform with the interface IF_STW_ITEM.

Retrieve item

```
DATA:
  lo_item  TYPE REF TO if_stw_item.

" iv_item_id is the ID of an item we have created before
lo_item = go_api->get_item( iv_item_id ).
```

Create comment

```
DATA:
  lo_item  TYPE REF TO if_stw_item.

" iv_item_id is the ID of an item we have created before
lo_item = go_api->get_item( iv_item_id ).

lo_item->create_comment( 'Everybody wants to go to heaven; but nobody wants to die.' ).
```

Create item

```
DATA:
  lv_item_id   TYPE if_stw_item=>tv_itm_id,
  lo_item      TYPE REF TO if_stw_item,
  lo_file_item TYPE REF TO if_stw_item,
  lo_activity  TYPE REF TO if_stw_activity.

" iv_activity_id is the ID of an activity we have created before
lo_activity = go_api->get_activity( iv_activity_id ).

" first we link the item to the activity
lo_file_item ?= lo_activity->create_item_instance(
                        iv_item_type = if_stw_item=>gc_item_type-file ).

" the second step puts the data into the item
" iv_filestream is the content of the file
lo_file_item->create_from_data(
                        iv_item_name         = 'Terminator'
                        iv_item_description  = 'Short movie'
                        iv_file_name         = 'hastalavista.mpg'
                        iv_file_content_type = 'video/mpg'
                        iv_file              = iv_filestream ).

" as usual: we have an ID
lv_item_id = lo_file_item->if_stw_item~get_id( ).

" ---------------------------------------------------------------
" now the same with a generic item
DATA:
  lt_attachment TYPE if_stw_item=>tt_attachment,
  ls_attachment LIKE LINE OF lt_attachment.

ls_attachment-file         = iv_filestream.
ls_attachment-part_name    = 'file'.
ls_attachment-filename     = 'hastalavista.mpg'.
ls_attachment-content_type = 'video/mpg'.
APPEND ls_attachment TO lt_attachment.

lv_string = |<file_item/>|.

lo_converter = cl_abap_conv_out_ce=>create( encoding = 'UTF-8' ).

lo_converter->convert( EXPORTING data = lv_string IMPORTING buffer = lv_xstring ).

" get an instance for creating generic item
lo_item ?= lo_activity->create_item_instance( iv_item_type = pitmtype ).

" create generic item
lo_item->create_from_data( iv_item_name        = 'Terminator'
                           iv_item_description = 'Short movie'
                           iv_item_detail_xml  = lv_xstring
                           it_attachment       = lt_attachment ).
```

Retrieve all items of an activity

```
DATA:
  Lt_item      TYPE if_stw_item=>tt_item,
  lo_activity  TYPE REF TO if_stw_activity.

" iv_activity_id is the ID of an activity we have created before
lo_activity = go_api->get_activity( iv_activity_id ).

lt_items = lo_activity->get_items( ).
```

Tasks

➔ The report RSTW_LIBRARY_TEST_TASK illustrates the options you have for this collaboration object and how to use it in your code.

Tasks are work items that can be assigned to one or more users, with a description, a due date, and a status. They live in the context of an activity.

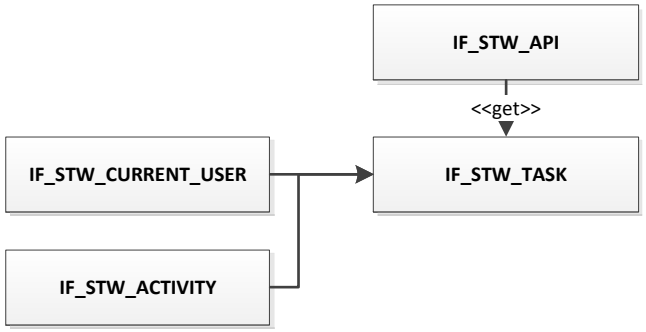


Figure 9: Main Object Relationships for Tasks

If you work with tasks, the following table shows the most common tasks you may want to perform:

Task	Remarks
Create task	Create a new task for an activity
Update task	Update a task
Set status	Set the status of a task
Delete task	Delete a task

A comprehensive overview can be found in the test report. The class CL_MAIN shows all the functions you can perform with the interface IF_STW_TASK.

Create task

```
DATA:
  lo_activity TYPE REF TO if_stw_activity,
  lv_task_id  TYPE stw_task_id.

" iv_activity_id is the ID of an activity we have created before
lo_activity = go_api->get_activity( iv_activity_id ).

" it_participants is the list of users that will participate in the task
lo_task = lo_activity->create_task(
                  iv_title       = 'Buy keyboard'
                  iv_description = 'Look for self-correcting keyboard'
                  iv_due_date    = '20991223'
                  it_participant = lt_participants ).

" the task ID can be used for later retrieval
lv_task_id = lo_task->get_id( ).
```

Update task

```
DATA:
  lo_task TYPE REF TO if_stw_task.

" iv_task_id is the ID of a task we have created before
lo_task = go_api->get_task( iv_task_id ).

lo_task->update(
                  iv_title       = 'Buy keyboard'
                  iv_description = 'Look for keyboard that can guess the next word'
                  iv_due_date    = '20991223' ).
```

Set status

```
DATA:
  lo_task TYPE REF TO if_stw_task.

" iv_task_id is the ID of a task we have created before
lo_task = go_api->get_task( iv_task_id ).

lo_task->change_status( iv_status = if_stw_task=>gc_status_inprogress ).
```

Delete task

```
DATA:
  lo_task TYPE REF TO if_stw_task.

" iv_task_id is the ID of a task we have created before
lo_task = go_api->get_task( iv_task_id ).

lo_task->delete( ).
```

Using the Log

There are situations in which even the most rock solid coding crashes - and they are always unexpected. In these cases, it is helpful if a log has been written in the background and you have the chance to read about what happened.

In cases like these, the SAIL API uses the Business Application Log (BAL) to write messages. The object registered in the BAL for the SAIL API is, not surprisingly, named 'SAIL', and the subobjects are 'PROC' (for common processing) and 'GWN' for processing within the context of business object notifications in the context of SAP NetWeaver Gateway.

The interface IF_CLB_LOG describes the log's capabilities.

The log always writes in the context of the platform type, the SAIL application and the SAIL application context – the application key. This key identifies each infrastructure layer instance, and this is where you can retrieve a log instance for your own purposes: From the single point of entry into the infrastructure layer, CL_CLB_LIB.

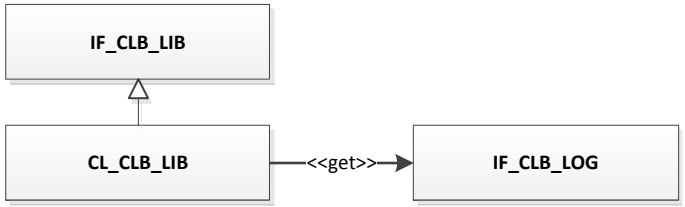


Figure 10: Main Object Relations for the Log

You can add exceptions, BAPI return tables, standard messages, and simple texts to the log as standard behavior.
In addition, it is possible to write trace and debug messages to the log. Debug messages are intended to be used for extended message output for error analysis. Trace messages are intended to be used during the development phase for additional output with regard to the program flow. Both message types should be used only in rare cases; therefore, a parameter has been introduced that has to be activated before such messages are written to the log. This parameter is CLB_LOG and appears in your parameter list in transaction **SU3** if properly activated. Possible values for this parameter are:

- 'A': Trace and debug messages are enabled
- 'T': Trace messages are enabled
- 'D': Debug messages are enabled

If you work with tasks the following table shows the most common tasks you may want to perform:

Task	Remarks
Write	Write a log entry
Activate additional log types	Activate trace and debug messages
Display	Display the log with standard transaction SLG1

Write

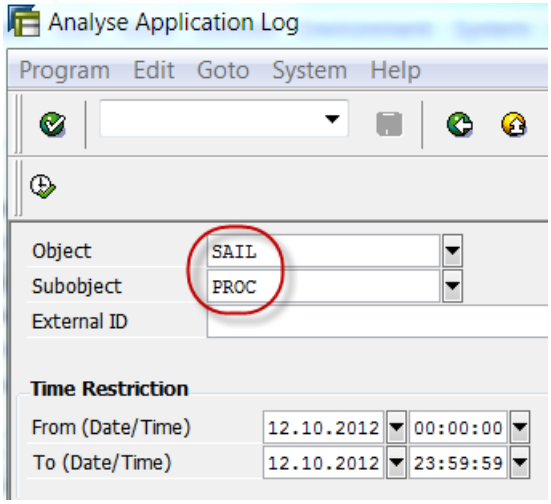
```
DATA:
  lo_log TYPE REF TO if_clb_log.

" iv_appl_key is our application key
lo_log = cl_clb_lib=>get_log( iv_appl_key ).

" write a simple text message
lo_log->add_text( iv_msg = 'If I agreed with you we'd both be wrong.' ).

" write a T100 message
MESSAGE s000(clb) WITH 'Dogs have owners'
                       'Cats have staff'
                  INTO lo_log->dummy.
lo_log->add_message( ).
```

Activate additional log types

```
" (de)activate debug and/or trace mode as requested
cl_clb_bal=>activate(
                   iv_debug_mode = 'X'
                   iv_trace_mode = 'X').
```

Display

```
" To display the log you can use transaction SLG1
```




```
" or you can get the log handle…
DATA:
  lv_bal_hndl TYPE balloghndl,
  lt_log_hndl TYPE bal_t_logh,
  lo_log      TYPE REF TO if_clb_log.

" iv_appl_key is our application key
lo_log = cl_clb_lib=>get_log( iv_appl_key ).

lv_bal_hndl = lo_log->get_handle( ).
CALL FUNCTION 'BAL_DSP_LOG_DISPLAY'
  EXPORTING
    I_t_log_handle = lt_log_hndl
  EXCEPTIONS
    OTHERS         = 0.
```

Tailoring the Process

BAdIs

User Mapping

In most cases, users IDs in the backend differ from the user IDs for the collaboration platform. Therefore, we need a mapping mechanism that maps both IDs. A BAdI serves this purpose:

Package:	S_STW_OL_COMMON
Enhancement spot:	STW_USER
BAdI definition:	STW_USER_EMAIL

The standard implementation maps the user's ID to his or her e-mail address in the backend. This requires that the e-mail addresses are maintained in the user data in the backend system.

Authorization

Virtually all of the methods provided by a collaboration platform are protected by an authorization that has to take place before the method can be executed. The type of authorization (SAIL API calls this the authentication context) depends on the endpoint of the collaboration platform because the collaboration platform determines the authorization process. The authentication context represents an abstraction of the methods actually used:

Context	Remark
APPLI	= Application context without user This authorization does not require a user. SAIL API implements this as a **2-legged OAuth (OAuth 1.0a)**.
APPUSR	= Application context with user This authorization requires a user, although the method calls protected by this authorization are executed outside the user's context. The user is required for audit purposes. SAIL API implements this as a **3-legged OAuth (OAuth 1.0a)**.
NONE	= No authorization There are some very rare method calls for which no authorization is required.
USER	= User context This authorization requires a user and grants access to all methods that may be executed in a single user's context. SAIL implements this via session ID through a **SAML2 assertion**.

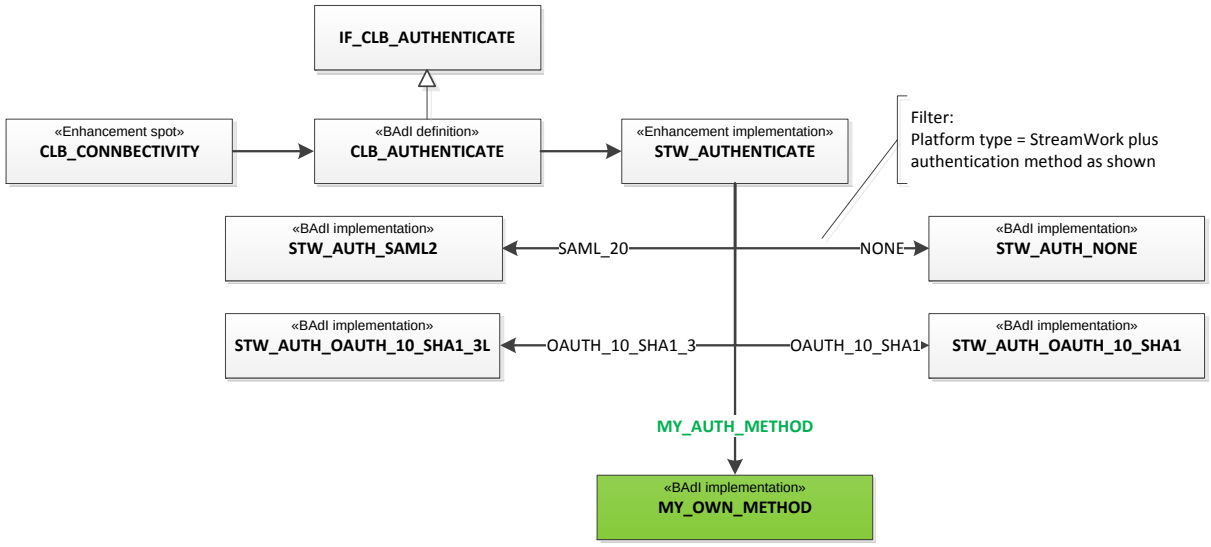
Each method implementing an endpoint of the collaboration platform (via interface IF_CLB_METHOD_PROCESS) carries the information about which authorization context it requires (as a return value from method GET_CONNECTION_DATA).

This authorization context (view CLBV_AUTH_CONT) is assigned a real authorization method using view CLBV_PLATF_AUTH. Common authorization methods are, for example, OAuth or basic authentication with plaintext.

The SAIL API comes with the authorization methods mentioned above. These are provided as BAdI implementations:

Package: S_CLB_CONNECT
Enhancement spot: CLB_CONNECTIVITY
BAdI definition: CLB_AUTHENTICATE

This is a filtered BAdI requiring the platform type and the authentication method as a filter.



So, if you want to provide your own authorization method as a replacement for one of the existing methods, you just have to implement a new BAdI with the new authorization method name as a filter and use view CLBV_PLATF_AUTH to assign your authorization method to an authentication context.
You may face this situation when you want to change the authentication flow or if you want to use the all new OAuth 2 protocol.

If, on the other hand, you want to implement a completely new authentication context, you can add it to the view CLBV_AUTH_CONT, but to make use of it, you will have to provide your own implementations of the endpoint methods that use this authentication context.
A new authentication context may be necessary when you want to connect to a new collaboration platform that requires an authentication context that is not available yet.

The authorization process was successful if the BAdI does not raise an exception.

There may be situations in which the standard processing does not fit the requirements. The standard execution of a REST call is shown on the next page.
Again, this may be necessary if you want to connect to a different collaboration platform or if you want to use a different authorization flow, to name just two instances.

In this case, you will have to replace the dispatcher class of the infrastructure layer, as it is responsible for the work flow of a REST call. How to do this is described later in this document.

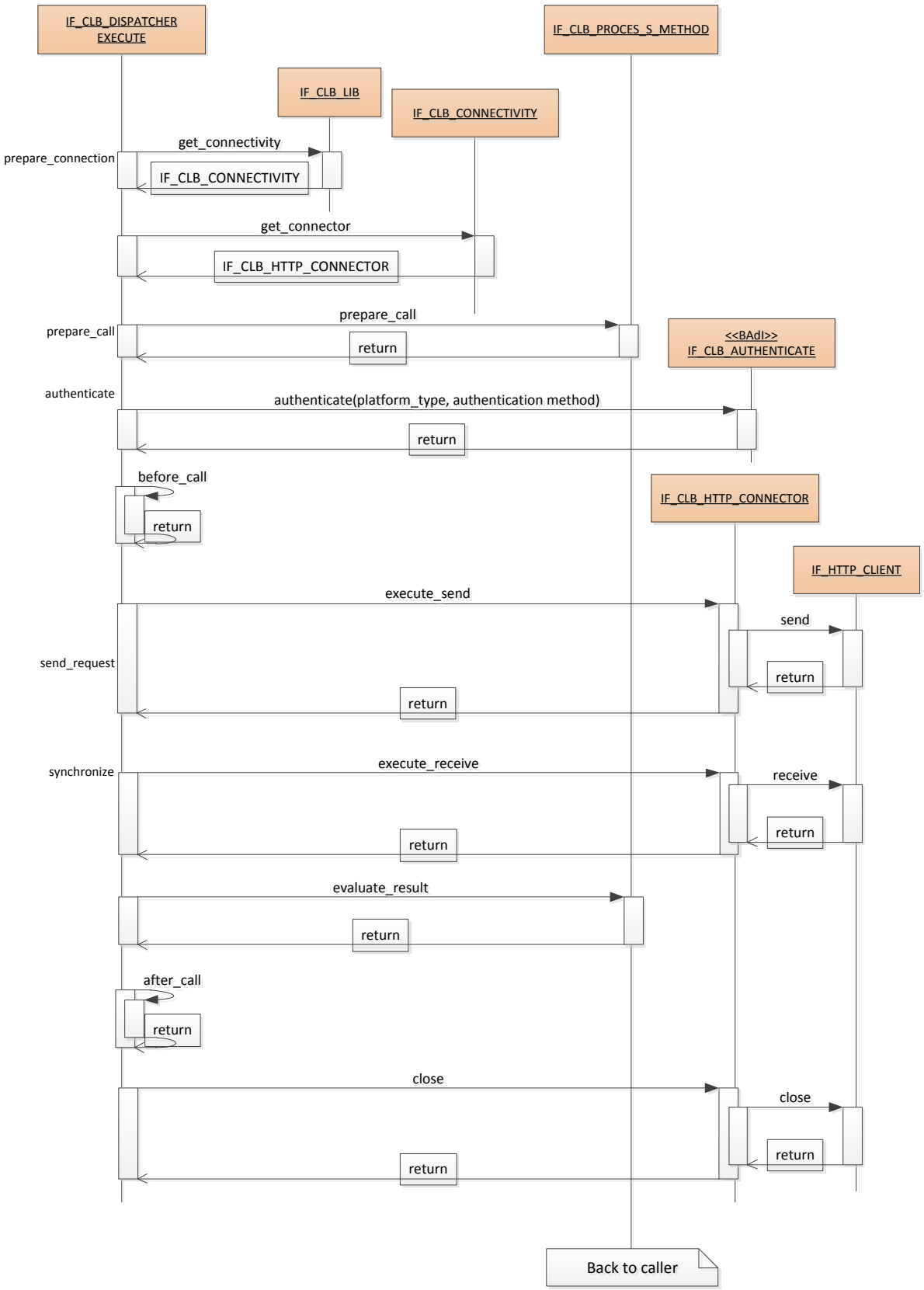


Figure 11: Basic Flow of a REST Call

Item Factory

From a coding point of view, an item basically consists of an XML stream. This XML stream transports both data and UI information between the collaboration platform and the backend. This XML stream can be seen as a header-data-structure. The header part is covered by interface IF_STW_ITEM and its basic implementation CL_STW_ITEM.
The data part is specific to the item, even for a particular version of an item. To be able to support all variations, a BAdI is provided:

Package: S_STW_OL_COMMON
Enhancement spot: STW_ITEM_FACTORY
BAdI definition: STW_ITEM_FACTORY2

SAIL provides three item types:

1. Generic items
   Just evaluates the header and carries the data as a blob; this is the fallback item if any other try fails.
2. Text items
   The data part of a text item is, not surprisingly, simple text.
3. File items
   The file item carries a file; not just the link but the content as well.

Additional item types require a new BAdI implementation. Again, this BAdI is a filtered BAdI, which has the application context as the filter. The application context, if you remember, is part of the application key. For historical reasons this filter is named **SCOPE**.
The BAdI implementation is responsible for creating an item instance of the item type in question, hence *FACTORY*:

Create an item type instance

```
…
  CASE iv_item_type.

    WHEN if_stw_item=>gc_item_type-text.
      CREATE OBJECT ro_item TYPE cl_stw_item_text
        EXPORTING
          io_controller = io_controller
          iv_item_type  = iv_item_type.

    WHEN if_stw_item=>gc_item_type-file.
      CREATE OBJECT ro_item TYPE cl_stw_item_file
        EXPORTING
          io_controller = io_controller
          iv_item_type  = iv_item_type.

    WHEN OTHERS.
      CREATE OBJECT ro_item TYPE cl_stw_item_generic
        EXPORTING
          io_controller = io_controller
          iv_item_type  = iv_item_type.
  ENDCASE.
…
```

Your custom class factory only needs to handle the additional/modified item types. If your implementation does not return an item object, the SAIL API takes over and creates an item instance on its own.
It should also be mentioned that you can use the application key (here SCOPE) from the very beginning and throughout all your work with the SAIL API.
However, you can also stay in the context that comes with SAIL. Just before you execute the code sequence that needs support from one of your own item types, make use of the SET_SCOPE method that comes with IF_STW_API. The value you set there will be used as the scope instead of the value provided in the application key.

Set scope for item type

```
"code sequence preparing the call
…
"now change the scope before we trigger the call
go_api = cl_stw_api_factory->set_scope( 'MY_FINE_SCOPE' ).
    " execute the method call
" and set the application context back
go_api = cl_stw_api_factory->set_scope( old_scope).
```

Method Substitution

You may have reasons for replacing an endpoint implementation (in SAIL called **method**) with your own version: For example, if the collaboration platform provides a new method that is not yet supported by the SAIL API, when a parameter is not supported in the endpoint implementation, or when you need to perform additional work on the method calls result – just to name a few.

The very first step, of course, is to check how you can fulfill your requirement the easiest way possible. You can

1. Inherit from an existing method
   This is the best choice if you just want to enhance the result of the method call.
2. Implement the interface IF_CLB_METHOD_PROCESS
   This is recommended if you want to have full control over the data sent to and received from the collaboration platform.
3. Use the generic method CL_CLB_METHOD_PROCESS
   This is a method you can use if you want to prepare/evaluate the data to/from the collaboration platform in a separate class hierarchy and just want SAIL to take care of the communication part.

To get one of these three up and running you will have to modify the Customizing.

To replace an existing implementation, you can use transaction **SM34** with the viewcluster CLBVC_PTYPE. Navigate to Collaboration: *Service Provider -> API Methods -> API Method Versions* and enter your new method implementation into the corresponding method.

Follow the same procedure if you want to add a new version of a method to support additional parameters. Just add a new method version and the corresponding class and you're done. Well, nearly done, as you will have noticed, the new version has to be made public. To do so, use view CLBV_PLATF_METH (transaction **SM30**) to indicate which version should be read for each method.

You need the same view cluster for new methods; start by adding a new API Method before you add a new version.

Okay, but what about the interface itself?
Well, this is a very simple interface:

1. GET_NAME returns the name of the method. This is used for enhancement of any error message.
2. GET_CONNECTION_DATA returns the values needed to collect the information you need in order to connect to the collaboration platform.
   You have to return the endpoint as defined by the collaboration platform (without the root URL, for example, ' /v1/activites/1234/owner'). The request method is required as the next value, that is PUT or GET or ... And the authentication context has to be specified.

GET_CONNECTION_DATA

```
DATA:
  lo_log TYPE REF TO if_clb_log.

" rs_data is our return structure
rs_data-auth_context = if_clb_constants=>gc_auth_context->user.
rs_data-endpoint = '/v1/activities/1234/owner'.
rs_data-request_method = if_clb_constants=>gc_request_method-put.
```

3. PREPARE_CALL is the last method called before the physical call to the collaboration platform is made. Here, you can manipulate the HTTP client (CL_HTTP_CLIENT) that is used to perform the call, for example, by setting the data into the object instance. You can also call a *simple transformation* to transform any ABAP structured data into its corresponding XML equivalent before you move the data to the HTTP client.

PREPARE_CALL

```
DATA:
  lv_xml TYPE string.

" ms_activity_data has been set from the outside
CALL TRANSFORMATION stw_act_change_owner_s
                    SOURCE owner_id = ms_activity_data
                    RESULT XML      = lv_xml.

" put the data into the HTTP client
io_connector->add_header_field(
                i_name  = 'Content-Type'
                i_value = 'application/xml' )..
io_connector->set_data( lv_xml ).
```

4. EVALUATE_RESULT is executed after the HTTP call to the collaboration platform returns without error code. The main task here is to transform the XML data into an ABAP representation.

EVALUATE_RESULT

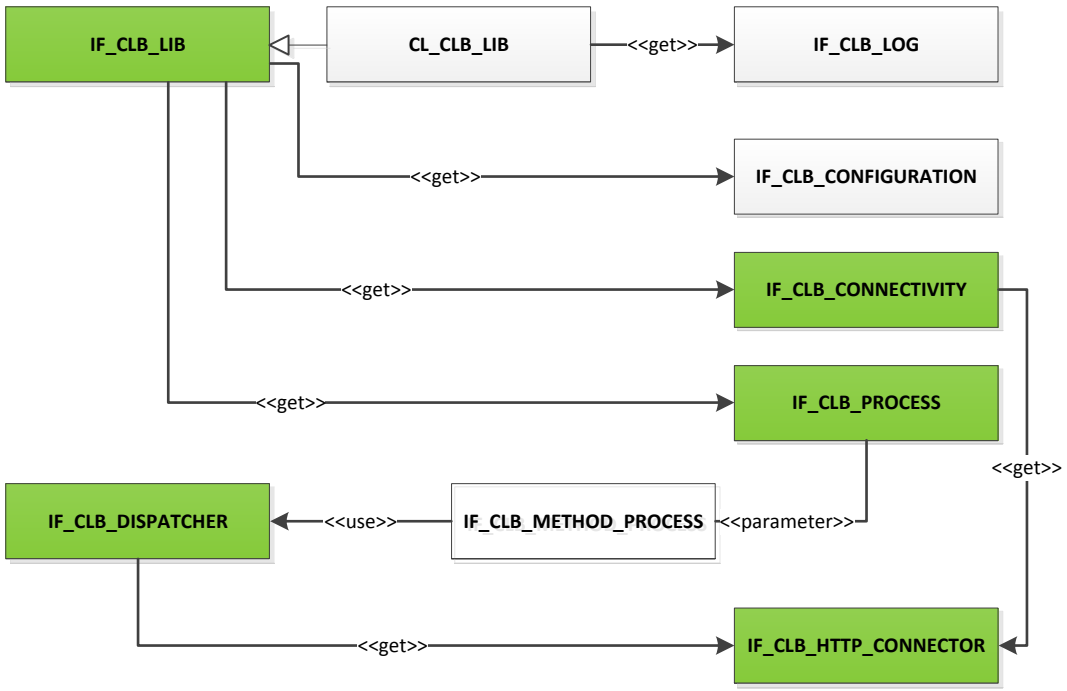
```
DATA:
  lx_exception            TYPE REF TO cx_clb_process_method,
  lx_transformation_error TYPE REF TO cx_clb_process_method,
  lt_Activity_data        TYPE if_stw_mth_common_types=>tt_activity_data.

IF iv_result_code <> '200'.
  lx_exception = cl_stw_mth_exception_factory=>s_get_instance( )
                   ->get_exception_instance(
                                        iv_result = iv_result
                                        iv_result_code = iv_result_code
                                        iv_result_message = iv_result_message ).
  RAISE EXCEPTION lx_exception.
ENDIF.

TRY.
    CALL TRANSFORMATION stw_act_get_activities_d
         SOURCE XML          iv_result
         RESULT activities   = lt_activity_data.

  CATCH cx_transformation_error INTO lx_transformation_error.
    RAISE EXCEPTION TYPE cx_clb_process_method
      EXPORTING
        previous = lx_transformation_error.
ENDTRY.
```

Replacing Infrastructure Interfaces

You can tailor the process even closer to your requirements by replacing some of the interfaces of the infrastructure layer. To do this, you should start transaction **SM30** with view CLBV_PTYPE_CF. The interfaces here are the heart and soul of the infrastructure layer. Just enter your own implementation of the interface that you want to replace and you are done.

- IF_CLB_LIB is the single point of entry into the infrastructure layer.
- IF_CLB_PROCESS controls the processing services of the infrastructure layer.
- IF_CLB_DISPATCHER controls the communication with the collaboration platform.
- IF_CLB_CONNECTIVITY is responsible for the physical connection to the collaboration platform. At the current stage it just provides access to the connector.
- IF_CLB_HTTP_CONNECTOR wraps the HTTP client into a convenience class.

The colored interfaces in the figure below can be replaced.



The most important class is the dispatcher class because it is responsible for executing the call.

Related Content

SAIL Configuration Guide	http://scn.sap.com/docs/DOC-24691

Copyright