

Oracle Health Insurance

Custom Development for

Oracle Health Insurance Back Office

version 1.19

Part number: F27899-01

March 18, 2020

Copyright © 2014, 2020, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this software or related documentation is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS

Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are “commercial computer software” or “commercial technical data” pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications which may create a risk of personal injury. If you use this software in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use of this software. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software in dangerous applications.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

This software and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Where an Oracle offering includes third party content or software, we may be required to include related notices. For information on third party notices and the software and related documentation in connection with which they need to be included, please contact the attorney from the Development and Strategic Initiatives Legal Group that supports the development team for the Oracle offering. Contact information can be found on the Attorney Contact Chart.

The information contained in this document is for informational sharing purposes only and should be considered in your capacity as a customer advisory board member or pursuant to your beta trial agreement only. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described in this document remains at the sole discretion of Oracle.

This document in any form, software or printed matter, contains proprietary information that is the exclusive property of Oracle. Your access to and use of this confidential material is subject to the terms and conditions of your Oracle Software License and Service Agreement, which has been executed and with which you agree to comply. This document and information contained herein may not be disclosed, copied, reproduced, or distributed to anyone outside Oracle without prior written consent of Oracle. This document is not part of your license agreement nor can it be incorporated into any contractual agreement with Oracle or its subsidiaries or affiliates.

CHANGE HISTORY

Release	Version	Changes
10.14.1.0.0	1.0	<ul style="list-style-type: none"> • Creation
10.14.1.0.0	1.1	<ul style="list-style-type: none"> • Minor adjustments
10.14.2.0.0	1.2	<ul style="list-style-type: none"> • Minor adjustments
10.15.1.0.0	1.3	<ul style="list-style-type: none"> • Minor adjustments 'Business Event Framework' • Minor adjustments 'Dynamic PLSQL'
10.15.3.0.0	1.4	<ul style="list-style-type: none"> • Minor adjustments 'Modification logging' • Minor adjustments 'Flex fields' • Added fileoutput in a specific character set to 'Custom batch scripts'
10.16.1.0.0	1.5	<ul style="list-style-type: none"> • No changes
10.16.2.0.0	1.6	<ul style="list-style-type: none"> • Added 'VPD implementation'
10.17.1.0.0	1.7	<ul style="list-style-type: none"> • Added table RBH_VDR_HISTORIE to VPD implementation. Added instructions for the use of custom developed PL/SQL objects in system views. • Minor adjustments on column bound checks, added the 'Direct?' checkbox
10.17.1.1.0	1.8	<ul style="list-style-type: none"> • Added 'Indexed access of Flex field Values'
10.17.2.0.0	1.9	<ul style="list-style-type: none"> • Added description for how to purge data from logging tables
10.17.2.2.0	1.10	<ul style="list-style-type: none"> • Adjustments for enhanced batch 'Tracing' functionality, a debug level and specific stored pl/sql units can be specified.
10.18.1.0.0	1.11	<ul style="list-style-type: none"> • Small adjustment regarding the reference of script OZG_DIRECT.grt.
10.18.1.2.0	1.12	<ul style="list-style-type: none"> • Corrected information about indexes not being present on change logging tables and enhanced description of open database.
10.18.2.0.0	1.13	<ul style="list-style-type: none"> • Removed references to C2B web services.
10.19.1.0.0	1.14	<ul style="list-style-type: none"> • No changes. Republished with different part nr.
10.19.1.2.0	1.15	<ul style="list-style-type: none"> • Added description how to put JMS messages on a database queue with JMS payload.
10.19.1.3.0	1.16	<ul style="list-style-type: none"> • Added Service Consumers as separate chapter
10.19.2.0.0	1.17	<ul style="list-style-type: none"> • Paragraph is added about using message handling calls in pl/sql definitions.
10.20.1.0.0	1.18	<ul style="list-style-type: none"> • Interface layout ALG_EVENT_INTERFACE_PCK modified • System and account mapping views grant structure changed • OHI_BATCH_ROLE documented
10.20.2.0.0	1.19	<ul style="list-style-type: none"> • Appendix H Service consumer examples added

Contents

1.	Introduction.....	1
1.1.	Audience.....	1
1.2.	Scope	1
1.3.	Documentation	2
1.4.	References.....	2
2.	Overview	3
2.1.	Business Rules.....	4
2.2.	An 'Open to Custom Code' Database.....	5
2.3.	Flex Fields	5
2.4.	Dynamic PL/SQL.....	6
2.5.	Business Event Framework.....	6
2.6.	Custom Batch Scripts	6
2.7.	HTTP Links	6
2.8.	OHI Back Office Business Services.....	7
2.9.	JMS Messaging.....	7
3.	An 'Open to Custom Code' database.....	8
3.1.	Tables and views	8
3.2.	Authorization.....	9
3.3.	VPD implementation	11
3.4.	Datamodel help	16
3.5.	Modification logging.....	18
3.6.	Tracing	25
4.	Flex Fields	27
4.1.	Concepts	27
4.2.	Flex Field Characteristics.....	28
4.3.	Flex Field definition	29
4.4.	Flex Field maintenance	31
4.5.	Indexed access of Flex Field values.....	31
5.	Dynamic PL/SQL.....	33
5.1.	Hooks for PL/SQL code	33
5.2.	Dynamic PLSQL Definition	33
5.3.	Column bound checks	36
5.4.	Writing custom code.....	42
6.	Business Event Framework.....	43
6.1.	Overview	43
6.2.	Signaling Events	43
6.3.	Responding to Events	44
6.4.	Combining Signaling and Response Types	44
6.5.	Framework Components.....	45
6.6.	Developing Your Own Business Events.....	50
6.7.	Processing Business Events.....	54
6.8.	Examples.....	55
6.9.	Trouble shooting the event framework.....	61

7.	Custom Batch Scripts	62
7.1.	Approach	62
7.2.	Batch user	63
7.3.	Registration	63
7.4.	Export script definition.....	66
7.5.	Generator: Bulk Processing Batch - Overview	66
7.6.	Generator: Bulk Processing Batch - Details.....	67
7.7.	Generator: CSV Export Batch.....	73
7.8.	Generator: CSV Import Batch	74
7.9.	Generator: XML Import batch	75
7.10.	Other output scripts	75
7.11.	Module installation script	75
7.12.	Creating output in a specific charset.....	75
8.	HTTP Links.....	76
8.1.	Configuration.....	76
9.	OHI Back Office Business Services.....	80
9.1.	Architecture.....	80
9.2.	Implementation.....	80
9.3.	Find, Get and Write Services.....	81
9.4.	Write Services.....	81
9.5.	Error Handling.....	85
10.	Service Consumers	86
10.1.	WSDL transformation.....	86
10.2.	Request and Response JMS payload queues	87
10.3.	Message Processing.....	87
10.4.	Custom calls on service consumers	89
11.	Custom Development Practices.....	90
11.1.	Create a custom schema	90
11.2.	Use an abstraction layer.....	90
11.3.	Define your transactions.....	91
11.4.	Locking	92
11.5.	Use Named Parameters	92
11.6.	Profile your code	93
11.7.	Close open cursors	93
11.8.	Coding Standards.....	93
12.	Deprecated Interfacing options	94
13.	Appendix A - Business event framework datamodel.....	1
14.	Appendix B - Business event interface ALG_EVENT_INTERFACE_PCK.....	4
15.	Appendix C - Tracing.....	7
15.1.	Activation	7
15.2.	How to Access Trace Logs.....	8
15.3.	Instrumentation of Custom Code.....	8
16.	Appendix D - What you should know about CDM RuleFrame.....	9
17.	Appendix E - Modification Mechanism for Policies and Relations	10
18.	Appendix F - Dynamic PL/SQL Types.....	11
19.	Appendix G - Using a JMS payload queue.....	13
19.1.	Starting points for the standard OHI queue with JMS payload.....	13
19.2.	Points of attention.....	14
19.3.	Interface package.....	14
19.4.	Enqueue example	16

20.	Appendix H - Service consumer example calls	18
20.1.	Check on COV	18
20.2.	Municipal administration (BRP/GBA).....	21

1. Introduction

Two of the strengths of the Oracle Health Insurance Back Office application are the possibilities to configure and customize the application.

Much of the behavior of the application can be influenced through configuration parameters.

If this is not enough, the application offers facilities to:

- Extend the standard data model with flex fields;
- Include custom developed code into the standard processes;
- Define and process custom-defined events;
- Develop custom applications to integrate with OHI Back Office through web services; or
- Develop custom PL/SQL code to directly access OHI Back Office data and implement customer specific functionalities (data processing processes, output producing processes, etc.)

The purpose of this document is to describe these options and help you decide how to customize OHI Back Office to your needs.

1.1. Audience

This document is primarily written as a technical reference for anyone who needs to integrate with OHI Back Office or to augment the standard process.

Since this document describes the various options to integrate with OHI Back Office it may also be of interest to application managers .

The document assumes familiarity with:

- Basic OHI Back Office functionality
- Relational database concepts
- SQL and PL/SQL
- Secure Design and Secure Coding principles

Also the document assumes access to the OHI Back Office Online Help.

1.2. Scope

Within scope:

- Best practices for custom development and customization
- SVL web services
- How to organize custom code

Out of scope:

- Configuration of OHI Back Office
- Localization (translation, application code for local jurisdiction).
- Post-processing XML
- Client applications

- Interfacing with BI
- Self-service applications
- Coding standards
- Version control for custom developed code
- How to set up a custom schema and access rights (for more information, see the technical manual Oracle Health Insurance Back Office - Object Authorization)

1.3. Documentation

You may find more detailed technical documentation (like this manual) at <http://docs.oracle.com>:

- Browse <http://docs.oracle.com>
- Navigate 'Industries > Insurance > Oracle Health Insurance Applications > Health Insurance Back Office'

Also use OHI Back Office Online Help, available through the Forms application but also separately accessible by an OHI Back Office specific URL, to get:

- Functional information about the topics
- The menu path of the mentioned screens in this document

1.4. References

DocRef	Document
Doc[1]	Back Office Service Layer Installation & Configuration Manual (docs.oracle.com)
Doc[2]	Back Office Service Layer User Manual (docs.oracle.com)
Doc[3]	Oracle Health Insurance Back Office Installation, Configuration & DBA Manual (docs.oracle.com)
Doc[4]	Oracle Health Insurance Back Office - Service Consumer Installation and Configuration Manual (docs.oracle.com)

2. Overview

OHI Back Office is an 'all in one' engine to process claims for health care payers. The health care is provided by providers who may be contracted beforehand by the health care payer.

In most cases, the insured members are policy holders for which they collect premiums. So product definition, health care and collection of premiums are core processes as well.

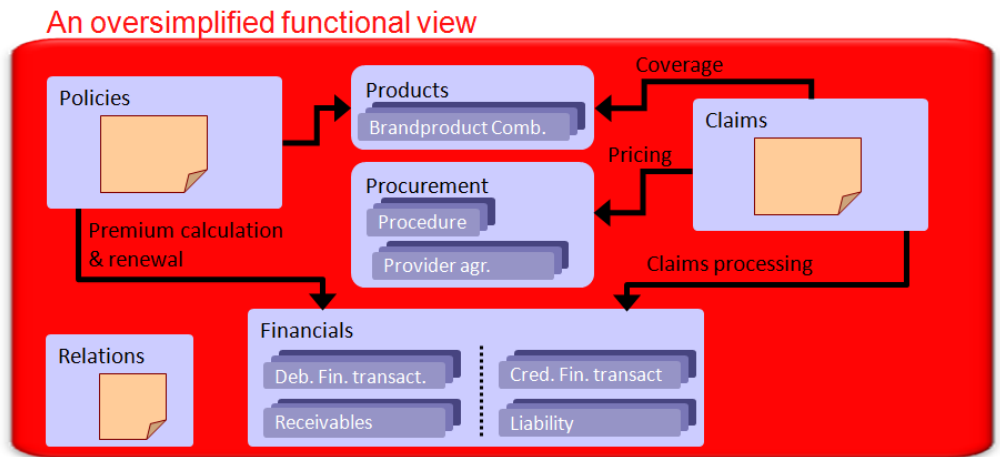


Figure 1: OHI Back Office 'all-in-one' core processes.

We consider OHI Back Office as an 'all-in-one' application, because all data, business rules and logic to support the core processes reside in a single database. This all-in-one approach allows maximum consistency for OHI Back Office data and is instrumental for achieving a high 'straight-through processing' (STP) ratio with a fairly limited use of resources in relation to the provided richness in functionality.

The OHI Back Office GUI application allows back office personnel to perform data entry, adjudication and run background processes. Much of the application logic for these tasks is implemented in the database.

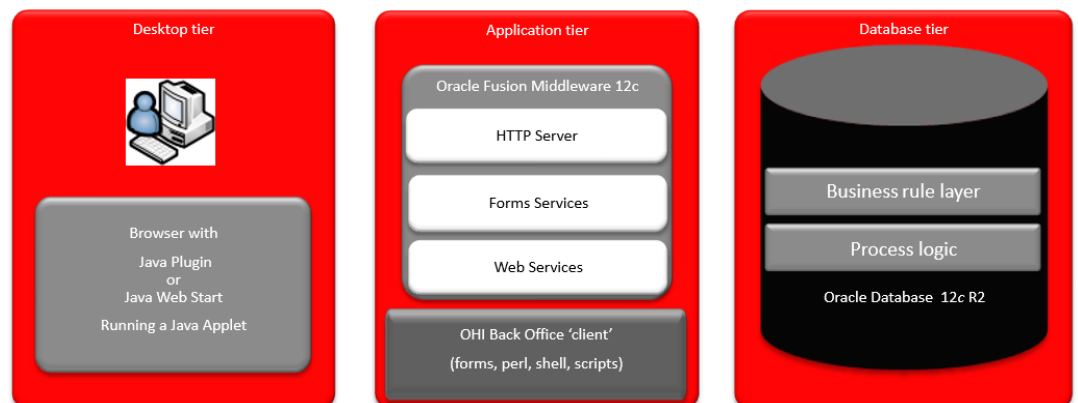


Figure 2: OHI Back Office GUI components

Although OHI Back Office is an 'all-in-one' application it is by no means a black box that cannot be altered. To work for a given customer OHI Back Office provides several possibilities for customization. To work in an environment with other applications, OHI Back Office provides various integration facilities. This chapter gives you an overview of these customization and integration facilities.

2.1. Business Rules

2.1.1. CDM ruleframe

Business rules are used to keep the OHI Back Office data in a consistent state for every DML operation. The OHI Back Office has many thousands of business rules which are implemented through an OHI Back Office tailored and optimized version of CDM RuleFrame. CDM RuleFrame is a framework originally developed for Oracle customers to develop their own custom Rule engine within an Oracle database.

The execution of the business rules is largely controlled by database triggers. With so many business rules these are bound to be many interdependencies, which means that business rules must always be adhered to and may never be disabled during use of the application. An exception may be well trained OHI Back Office personnel implementing customer specific conversions during a maintenance period.

Much of the processing of the business rules takes place when a transaction is committed to the database. This means that transactions should be defined as logical units to prevent that (seemingly) unrelated business rules are triggered by closing a transaction.

Because of their implementation in the database, business rules are tightly integrated with the data. This results in optimum performance. And, since the business rules apply to every DML operation on OHI Back Office objects, we created an 'Open Database' in respect to developing interfaces and custom code as you can access the database with (virtually) any tool that provides database access. Of course this openness and flexibility comes with great responsibility how to make use of it.

Finally, business rules help to avoid redundant code. Since they are always automatically applied, most application code does not need to be aware of the underlying business rules. This makes it easier for us to maintain and control our application code.

It also makes it viable for you to create custom code for OHI Back Office.

2.1.2. How to determine the set of rules on a table

There are many types of business rules like attribute rules, tuple rules, entity rules, inter entity rules which can also be divided in dynamic rules and static rules. There are several ways to determine the set of business rules on a given table.

- Check the constraint definitions in the data dictionary
- Look in alg#business_rules for the more complex static business rules, these are rules that can be validated at any moment in time
- Query on the system messages, business rule messages have the following format:
 - Start with a three letter application system prefix
 - Followed by _BR_
 - Followed by the three letter table alias which can be found in the column table_alias in ALG_TABELLEN for the given tablename
 - Followed by a three digit sequence number.
 - Followed by the classification of the rule.
 - `SELECT *`
`FROM ALG_SYSTEEM_MELDINGEN`
`WHERE CODE like '%_BR_%' escape '\'`

- Use the Online Help which provides an overview per table of the business rules. This information can also be generated for a specific release which is discussed later on in this document.

2.2. An 'Open to Custom Code' Database

You may directly access the OHI Back Office database because the business rules, implemented in the database itself, work as a protective shield.

You can either use the PL/SQL API packages direct access to the tables to access the OHI Back Office data, provided that you:

- Do not use the OHI Back Office schema owner to connect to the database (use a custom schema instead; in fact, your DBA should never grant access and lock this account when not in use for applying a new OHI patch/update)
- You should never (be able to) change the table definitions, types, packages or other database objects owned by the OHI Back Office schema as this means you might lose your right on OHI support
- Do never create custom grants from OHI Back Office database objects to your own schema.
- Do never disable business rules (you should never receive the privileges to do so)
- Do not insert, update or delete records from technical tables, including log and audit tables (you should, again, never receive the privileges to do so)

You can use and access every object which has been granted to your account without violating the above rules provided these are granted in the official supported way (see the Object Authorization manual) and you have not received any system privilege that provides more privileges than allowed.

You can use any tool or interface to access the OHI Back Office as long as the underlying interface is supported by Oracle (for example OCI, ODBC, JDBC, SQLNet, ODP.net).

2.3. Flex Fields

As a standard solution for healthcare payers, the OHI Back Office data model cannot be altered. However, the OHI Back Office data model can be extended with so-called flex fields to register additional customer-specific data, for example to record when members had their last health check or to register parameters for interfacing with providers.

These flex fields have the following benefits:

- The core data model stays unchanged, which means that OHI Back Office can support the application as sold to the customer.
- The customer does not need to create a secondary data model, the data of which must be kept in sync with the OHI Back Office database.

Historically, flex fields were used for claims line processing and to add extra data to policies and relation data. Soon after, flex field implementations were created for other tables and can now be commonly used on 'functional tables' (what this means is discussed in the next chapter).

Flex field support might be further extended in future releases.

2.4. Dynamic PL/SQL

OHI Back Office's support for dynamic PL/SQL is a powerful mechanism which allows the customer to add customer-specific code to the core OHI Back Office product.

OHI Back Office has defined several 'hooks' in OHI Back Office where custom code can be added:

- Data entry validation (specific as well as generic for all functional tables and their columns)
- Address entry and formatting.
- Trigger conditions for events using the 'business event framework'.
- Core processes such as claims processing, policy collection and payment processing.

Apart from adding a high level flexibility to the customer our support of dynamic PL/SQL helps to keep the amount of code in the standard product under control.

2.5. Business Event Framework

With the Business Event Framework you can define customer-specific events and event handlers.

These events can be time-based, triggered by a change or created by your custom-designed detection mechanism.

The handlers can be run near real-time in the background, asynchronously, or in batch processing mode at scheduled moments and are implemented through custom PL/SQL code.

2.6. Custom Batch Scripts

OHI Back Office allows you to create your own batch processing scripts and run them with the standard batch scheduler.

You are advised to use the built-in OHI Back Office script generator to generate a framework to control the execution of your custom code, especially if you expect to be processing large data volumes. The generated framework helps to process large volumes in manageable chunks, divide the work over parallel sub processes, and provide the same kind of feedback as the standard scripts.

2.7. HTTP Links

HTTP Links allow users of the OHI Back Office GUI Application to view or process related data in an external application. For each HTTP link you can configure a HTTP request template to the target application and the OHI screens which will display the HTTP link.



Figure 3: Toolbar with HTTP links

At runtime the HTTP link will open an extra browser window to send a HTTP request to the target application, substituting placeholders with runtime values derived from the current OHI screen.

Note that the target application must be an HTTP application! And when the accessed data should not be generally available make sure you implement some form of a Single Sign On access to prevent a user needs to supply a username and password for each callout.

2.8. OHI Back Office Business Services

The OHI Back Office Business Services add integration facilities for use in a service oriented environment. The business services are part of a service layer for retrieving and updating core OHI Back Office data.

The 'Find' and 'Get' services are used to retrieve data from OHI Back Office. 'Write' services are used to store data in the OHI Back Office database.

OHI Back Office Business Services are implemented as PL/SQL services and made available through synchronous SOAP/HTTP web services.

Note that the OHI Back Office Business Services are technically not related to the older OHI Connect2BackOffice (C2B) services. Although a license for that product historically has given you the right to use these Business Services.

2.9. JMS Messaging

The OHI Back Office application does come with a standard Oracle Advanced Queueing queue with a JMS payload implementation. This is a queue meant for generic use to offer out of the box messaging functionality when needed for publishing events to the 'world outside of OHI Back Office'.

As this queue is general purpose it is not optimized for specialized high frequency messaging tasks. But you can take the delivered queue implementation as an example and implement a customer specific tailored queue in a custom scheme.

For more information regarding this generic please see the [JMS payload queue Appendix](#).

3. An 'Open to Custom Code' database

You may directly access the OHI Back Office database because the business rules, implemented in the database itself, work as a protective shield.

You may use DML, PL/SQL API packages, or database views to access the OHI Back Office data, provided that you (see the previous chapter):

- Do not use the OHI Back Office schema owner to connect to the database (use a custom schema instead);
- Never change the table definitions, types, packages or other database objects owned by the OHI Back Office schema;
- Never create custom grants from OHI Back Office database objects to your own schema;
- Never disable business rules;
- Do not insert, update or delete records from technical tables (including log and audit tables).

This means you can and may use and access every object which has been granted to you by the application defined grant mechanism without violating the above rules.

You can use any tool or interface to access the OHI Back Office as long as the underlying interface is supported by Oracle (for example OCI, ODBC, JDBC, SQLNet, ODP.net).

The above means that the use of the OHI application is not supported and not certified when you do not adhere to these restrictions.

Effectively it is prohibited to:

- Execute any DDL that alters or influences the structure or behavior of OHI provided database objects within the OHI maintained schemas. So you may not drop, add, change, disable, enable, when applicable, tables, columns, indexes, pl/sql code, type definitions, etc. as this is all done by Oracle Health Insurance (OHI) delivered scripts. Only when instructed by OHI personnel an exception on this is allowed.
- Add, delete or change object privileges (grants) on the OHI provided objects or use system privileges to circumvent this object privileges.

It is allowed though to use DDL to reorganize the storage of data objects, so you may move and for example increase or decrease storage properties like `intrans` or `pctfree` or rebuild indexes. Provided that this may impact the use of OHI and you should prevent negative consequences and make sure a maintenance window is used for such actions when this is done offline.

3.1. Tables and views

3.1.1. Tables

These are the different table types in OHI Back Office:

- 'Functional' tables
 - All DML operations permitted
 - organized by functional area (aka sub system) like ALG, FSA, GEB or VER

- Technical tables: internal use only by OHI Back Office
- Logging tables:
 - used for tracking changes to the functional tables.
 - No DML operations permitted

The following naming convention is used to recognize each table type:

Type	Name format	DML?	Characteristics
Functional	<SUB SYSTEM>_<NAME>	Yes	<ul style="list-style-type: none"> • Functional • Rule layer • Column level grants
Logging	<SUB SYSTEM>\$<NAME>	No	<ul style="list-style-type: none"> ▪ Data changes ▪ On demand
Technical	<SUB SYSTEM>#<NAME>	No	<ul style="list-style-type: none"> • Process

3.1.2. Views

A similar naming convention is used for the different OHI Back Office view types:

Type	Name format	Characteristics
System	ALG_<NAME>_SVW	<ul style="list-style-type: none"> • Fixed column list • Customizable from and where clause
Financial	FVS_<NAME>_Sn_VW	<ul style="list-style-type: none"> ▪ For determination of General Ledger data ▪ Per financial fact type ▪ At least three fixed mandatory columns
English	<SUB SYSTEM>#<NAME>_	<ul style="list-style-type: none"> ▪ 1:1 view on functional table ▪ International (English) name and column names
English	<SUB SYSTEM>#<TABLENAME>\$	<ul style="list-style-type: none"> ▪ 1:1 view of logging table ▪ International (English) name and column names
Translation	<SUB SYSTEM>_<ALIAS>_VW	<ul style="list-style-type: none"> ▪ Translated values ▪ Public synonym to underlying table



Tip: Query the ALG_TABELLEN table to find the international view name for each table.

3.2. Authorization

Authorization and access in OHI Back Office is handled on three different levels

1. Database level

Table level select and delete grants in combination with column level insert and update grants provide a fine grained authorization implementation for changing data in the database of OHI Back Office.

A Virtual Private Database (VPD) OHI specific implementation can be activated to hide sensitive column data for users that are not privileged to access that data for specific records.

2. Application level

Application roles can be used to group a set of modules and provide write and/or read access to this set of modules by assigning the application role to

a user account.

Multiple application roles can be assigned to one account.

3. Business level

Per user the access to a financial unit or administration unit can be granted.

Per role the access to a brand, broker or group contract can be granted.



Note: The application level and business level access is only enforced through the Forms application.

3.2.1. Roles

Within OHI Back Office several database roles are present:

Name	Characteristics
OZG_ROL	<ul style="list-style-type: none"> ▪ Secure application role for user interface ▪ May not be directly granted ▪ For internal use only ▪ OHI_ROLE_ALL is granted to OZG_ROL
OZG_ROLE_ALL	<ul style="list-style-type: none"> ▪ Rights to all objects needed for executing OHI BO ▪ May not be directly granted. ▪ Accounts /GUI users get temporary elevation when needed (arranged by application). ▪ For internal use only
OZG_ROL_DIRECT	<ul style="list-style-type: none"> ▪ Rights directly granted to account ▪ Only used for custom interfaces or custom code accounts ▪ Not to be used for normal / GUI users
OZG_ROL_SELECT	<ul style="list-style-type: none"> ▪ Read only/query rights ▪ Not used by the application
OHI_ROLE_BATCH	<ul style="list-style-type: none"> ▪ Only used by the batch account to limit the privileges for that account

3.2.2. Accounts

Each OHI Back Office environment has a set of accounts:

Name	Characteristics
OZG_OWNER	<ul style="list-style-type: none"> • Owner of all OHI BO objects (name may differ per customer)
OHI_VIEW_OWNER	<ul style="list-style-type: none"> ▪ Owner of duplicated OHI view definitions needed for the VPD implementation
(GUI) USER	<ul style="list-style-type: none"> ▪ Per user ▪ Elevated to internal OZG_ROL when using forms application
BATCH	<ul style="list-style-type: none"> ▪ Used by the batch scheduler (name may differ per customer) ▪ Uses OHI_ROLE_BATCH
SVL_USER	<ul style="list-style-type: none"> ▪ User used for the service layer webservises
SVS_OWNER	<ul style="list-style-type: none"> ▪ Used for custom code development ▪ Should receive same grants as for OZG_ROL_DIRECT for stored code ▪ It is possible to create more than one custom development schema
OHI_DPS_USER	<ul style="list-style-type: none"> • Used when executing dynamic sql from OHI • Has the same grants as OZG_ROL_DIRECT • Contains the 'system' and 'financial translation/account mapping' views which may contain customer specific and configurable query text
Query accounts	<ul style="list-style-type: none"> ▪ Elevated to OZG_ROL_SELECT

3.3. VPD implementation

3.3.1. Introduction

Oracle Virtual Private Database (VPD) is a database feature that enables shielding sensitive data at the database level. In OHI Back Office VPD has been implemented for the tables listed below:

VPD tables
INT_POLISBERICHTEN
INT_VERZEKERDEN
RBH_BURG_STAAT_REGISTRATIES
RBH_EIGEN_REKENINGEN
RBH_RELATIES
RBH_RELATIE_ADRESSEN
RBH_VDR_HISTORIE
VER_AFGEGEVEN_ZORGPASSEN
VER_EESSI_BERICHTEN

For these tables series of columns are identified as 'sensitive' as they contain data for which restricted access can be implemented.

A reason for restricting access to columns is that they contain data that may be used to identify a person on who the record and its associated records apply (like name, date of birth, social security nr, phone nr, email address, physical address).

Within OHI a VPD implementation has been realized which nullifies column values for sensitive columns in records for which a user has no access. The record itself will always be returned.

When using any of these table names in custom code be aware of the following implications:

- Individual columns in specific records may be nullified as a result of VPD setup. When this happens the OHI Back Office user executing the code is not allowed to see this data for privacy reasons.
- In addition, these records may not allowed to be updated or deleted by specific users. Doing so will result in an error message: ALG187: "Modification is not allowed".

When for example certain custom batch processing functions require access to all data these should be executed using an OHI Back Office user that is authorized to query and modify this data.

In the following paragraphs the building blocks of the VPD implementation will be discussed including the implications for custom development.

3.3.2. VPD policies

The VPD policy is the definition in the database that is used to determine what data should be shielded and from which (database) users.

There are two different policy *types*. For tables that contain sensitive information in a column that is also part of an index, *rowlevel* policies are used. For tables that do not contain sensitive information that is part of an index *column-level* policies are used.

Column level policies apply to this subset of VPD tables and they are used to shield the data in specific columns based on the access privilege:

VPD tables with column level policy
INT_POLISBERICHTEN

VPD tables with column level policy
INT_VERZEKERDEN
RBH_VDR_HISTORIE
VER_AFGEGEVEN_ZORGPASSEN
VER_EESSI_BERICHTEN

Row level policies apply to this subset of VPD tables:

VPD tables with row level policy
RBH_BURG_STAAT_REGISTRATIES
RBH_EIGEN_REKENINGEN
RBH_RELATIES
RBH_RELATIE_ADRESSEN

- Because of the rowlevel policy type the use of the 'for update of <column specification>' clause in a SQL statement or cursor definition will result in an error if any column in the designated <column specification> contains sensitive information.

As the column specification is only documentative this can be easily resolved by changing the column specification of the for update clause to for example the primary key column or any other column(s) not containing sensitive information.

- As for OHI application code it is important to always retrieve a record even when the calling user has no access to the sensitive column values in a row the rowlevel policy is transformed to a column-level policy for the calling code. How this is achieved is described in the next paragraphs.

3.3.3. The VPD (& multi-language translation) view layer

The OHI Back Office implementation of VPD uses a 'view layer' to enable data hiding at column level without disabling indexed access of the table. In this way records that contain unauthorized column values can still be returned (as with a row-level policy such a record would not be returned). This is implemented by the so-called VPD view on top of a table that contains sensitive column data. This is done in a quite similar way as the already existing views for tables that contain translatable columns. This means that the implications for these VPD views also apply for the already existing views containing multi-language translatable columns (a limited set of mostly configuration setup tables).

Instead of directly access to a VPD eligible table access (select/insert/update/delete) to the associate VPD view is granted for custom development.

Any (PL/)SQL reference to a VPD table is redirected to the corresponding VPD view by means of a public synonym. This means that when selecting records from the table using the table name the user is actually retrieving data from a view that filters any sensitive data in accordance with the VPD setup. Prefixing the table name with the table owner does not work and is not allowed as access to the table is not granted.

For the same reason any DML against the table is actually performed on a view. The DML is passed on to the table using trigger code implemented by an 'instead of trigger' on the view. This trigger code checks if the user executing the DML is entitled to perform DML on the records involved based on the VPD role based data authorization setup.

The view layer has the following implications for custom development when using a VPD table:

- Use of an application owner prefix to specify a VPD table (e.g. OZG_OWNER.RBH_RELATIES) is not supported. This will not work because no privileges are granted for a VPD table. Any existing grants will always be revoked during OHI release installations using the default setting of the object privilege script. This even when 'enforce grant security' is not activated.
- Use of the RETURNING INTO clause is not supported for VPD tables because at runtime a view having an instead-of trigger is referenced instead of the table. This construct should be recoded manually in a different way to prevent the error message: ORA-22816: "Unsupported feature with RETURNING clause".
- The same restriction also applies for tables that can be translated, although custom DML against these tables will not occur frequently.
- Inserts in custom code that specify an explicit null value for a column that has a default value specification will still implement the default value as the instead-of trigger does not make any distinction between non specified or null specified column values in the insert on the view.

The VPD and translation types of tables can be identified using the query:

```
select tab.naam
from   alg_tabellen_tab
where  tab.vertaalview_naam is not null;
```

There are multiple methods to retrieve server derived data without the use of the RETURNING INTO clause. In most cases the generated primary key (ID column) is needed after creation of a new record using the INSERT statement. This value is generated by a sequence. For returning this value one of the following two approaches may be used.

1. The first approach is to generate the sequence value before the actual INSERT using the `nextval` function and to pass this value in the INSERT statement as the value for the ID column. To enable this the ID column of all VPD- and translatable views is insert-allowed. In addition the generation of a new relation number (NR column) is available for custom development by means of `API_RBH_UTIL_PCK.get_next_rel_nr`. Because multiple database sessions may be generating sequence values at the same time, using the current value of a sequence is not reliable for this goal. The `currval` function may return a value generated in another session and should *not* be used.
2. A second approach is to use a separate SELECT statement to retrieve the generated sequence value from the database. The record that was inserted should be identified using a unique key other than the ID column. If this is not available the first approach should be used. Please note that columns in a unique key may contain NULL values as they may be optional.

In some cases also other column values will be needed as a returning value. This can be due to server derived column values or changed data as a result of business rules that have fired.

3.3.4. Separate view owner schema

In the previous paragraph the use of a public synonym has been introduced that redirects access of the VPD table to the VPD view. To shield private data effectively this translation must take place any time a VPD table is referenced.

For this reason all views are duplicated to a separate OHI_VIEW_OWNER schema. This excludes system views and account mapping views (in fact 'data transformation

views', not to confuse them with the multi-language translation) views which can be defined and created from within the application. Also views that define the VPD and multi-language translation view layer itself are excluded.

This ensures that any reference to a VPD table from within these views will be translated to the corresponding VPD view using the public synonym.

Instead of the original view residing in the application owner schema the duplicated view in the view owner schema (OHI_VIEW_OWNER) is granted for custom development.

3.3.5. System and account mapping views

When a custom developed object (for example a package or table) is used in a system- or account mapping view, upon creation the SELECT privilege on this view will be granted to the roles OHI_ROLE_ALL and OZG_ROL_SELECT. Therefore, to prevent error ORA-01720: "Grant option does not exist for <OBJECT>", this object must be granted to OHI_DPS_USER using the 'WITH GRANT OPTION' clause.

3.3.6. Invokers rights

To redirect access of the VPD table to the VPD view in PL/SQL (using the public synonym) almost all OHI application standard PL/SQL is compiled as invokers rights unit. This also means that this PL/SQL will be executed using the privileges of the invoker (caller) instead of the definer (the owner of the program unit).

More information on invoker versus definer rights can be found in the Oracle Database documentation.

When using definer rights for a program unit underlying called objects in the same schema do not need to be granted explicitly as they are owned by the same database schema. Using invoker rights however privileges for these objects may be missing if these are not also granted to that caller (which will for sure be the case as not all objects may be accessed by the caller).

To resolve this situation the database feature Code Based Access Control (CBAC) is used. The privileges needed to access underlying objects are granted to the database role OHI_ROLE_ALL. This database role is granted to all OHI Back Office PL/SQL units. As a result PL/SQL units that have been granted for custom development will execute using the privileges of this role.

Be aware of the following implication related to the use of CBAC grants:

- Manually recreating OHI Back Office PL/SQL objects will cause the CBAC grant to be revoked. Although this action is not allowed, it may cause errors in the OHI Back Office application when this is accidentally done by a DBA or developer which has received access to the OHI object owner account. As a precaution measure do not open OHI object code with a privileged account that can change these objects. If this situation occurs, use the OHIPATCH menu to re-issue the CBAC grants using OHIPATCH step 120.

More information on CBAC can be found in the Oracle Database documentation.

New and existing custom developed batches generated using the batch framework (by means of SYS_GEN_PCK) will result in definer right PL/SQL packages. This means that the objects referenced by these custom batches that are also owned by the custom database schema do not need to be granted separately. To be more clear, for custom code there is no need to use invokers rights pl/sql objects towards OHI objects as the definer/owner of such code already has limited privileged access to OHI objects.

3.3.7. Performance aspects

Custom PL/SQL code and separate SQL queries will be using the VPD view layer when accessing the VPD tables listed previously. This will result in a higher resource usage on the database server because:

- Authorization should be checked for each single record to see whether the OHI Back Office user involved is allowed to see and modify the data. In most cases this will not be noticed because it is just a small part of the application logic as a whole but in some situations this may result in a slower response to database requests.
- The SQL query optimizer that determines the best access path will possibly follow a different route than before (and potential inefficient route), which may result in slower response times. Although the focus during development was to minimize this effect, this may be noticeable at times.

There is one known issue regarding the use of the VPD view layer: The optimizer does not seem to consider the use of function based indexes when this is appropriate without the use of an additional (dummy) condition.

An example:

The VPD table RBH_RELATIES contains a number of function based indexes, such as the ones defined on the concatenation of column values of UPPER_NAAM and UPPER_NAAM2. This is index RBH_REL_15 defined as RHB_REL_INDEX_CONCAT_NAAM(UPPER_NAAM,UPPER_NAAM_2). This index is used frequently within OHI application code when querying for a record in RBH_RELATIES having a specific last name. Because of the VPD view layer this function based index will not be used by default, which might result in poor performance.

By adding a dummy condition on the required column UPPER_NAAM the SQL optimizer is able to 'find' (use) the index again. The dummy condition to add in this situation would be UPPER_NAAM = UPPER_NAAM. The fact that this is a *required* column is essential for being able to added a dummy equals condition like this one, as with an option column certain records would possibly be excluded from the result set because of NULL values. In such a situation a different dummy condition needs to be applied if deemed necessary.

3.3.8. Additional measures taken

If needed, the column sequence of a table definition is automatically resequenced upon installation of an OHI release. This ensures that the column sequence of the VPD table in a %rowtype PL/SQL variable definition which is used during compilation time is equal to the column sequence of the VPD view based %rowtype variable which is used at runtime.

This enforcing of a standardized column order will prevent rowtype mismatches when using the SELECT * construct to fetch a row having the %rowtype of the view into a %rowtype variable having the %rowtype of the corresponding VPD table.

Because of the column resequence executed during installation this aspect has no consequences for custom development but you may notice it when describing a table. Columns that were recently added no longer appear at the end of the table definition.

As this resequence is only a dictionary change the actual order of columns as stored within a block is not changed and may differ from the definition order.

3.4. Datamodel help

One of the difficulties when starting to write custom code for OHI is the huge number of tables and relations between the tables. However, there are a few pointers to help you find your way around the data model.

3.4.1. Forms frontend

The first help for the programmer are the OHI screens themselves. If there is an item in a screen for which you want to know the location in the database, i.e. the table and the column, move the cursor to this field and select information from the help menu (Menu option: Help → Information), see Figure 4

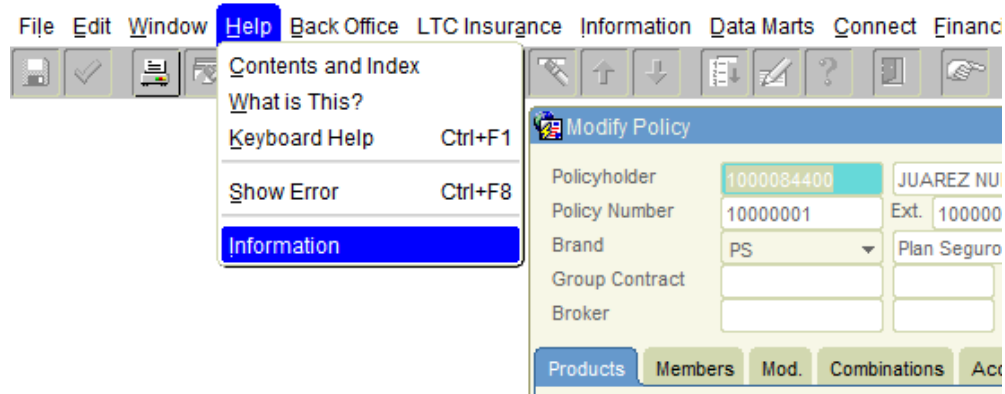


Figure 4: Information in the forms frontend

In this example the focus is on the policy holder number in the add policy screen, which means that you will get specific information about this field.

In figure x we called up information about a policy product start date. The information screen shows information about the screen (e.g. its module code ZRG2202F) and the database. But most importantly it shows the table name and the column name for the start date, both in Dutch and in English. As a bonus, it shows the record ID of the data that is currently shown. This might very useful when debugging some code.

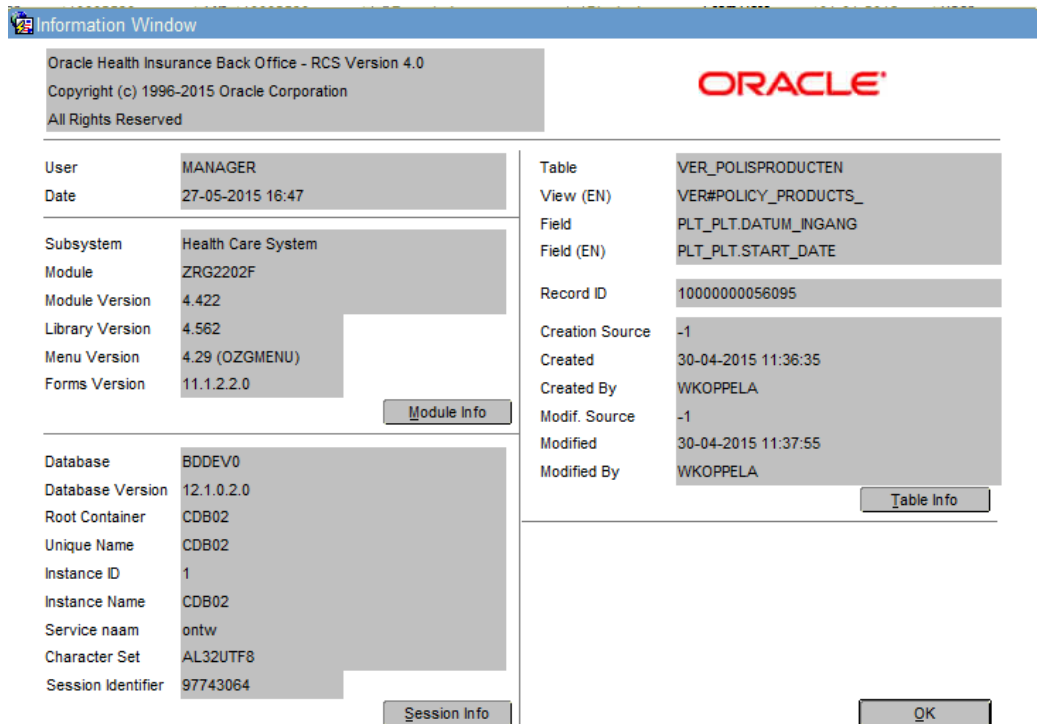


Figure 5: Detailed information per forms field

3.4.2. HTML overview

A more detailed picture can be created by using a procedure from OHI that creates a description of the tables and their columns. The package `SYS_GEN_PCK` contains the procedure `write_html_datamodel` with the following signature:

```
procedure write_html_datamodel
( pi_file_location in varchar2
);
```

The parameter `pi_file_location` must refer to a database directory in which a zip file with HTML files will be created:

`DatamodelOHI_<release_number>.zip`

So calling the procedure on OHI Back Office database for release 10.14.1.0.0 in a way like:

```
begin
  sys_gen_pck.write_html_datamodel
  ( pi_file_location => 'OZG_TMP'
  );
end;
/
```

will generate the file `DatamodelOHI_10.14.1.0.0.zip`. The zip file contains hundreds of HTML pages showing some simple but often needed information about the OHI tables that a programmer may access.

The pages are accessible through 5 different indexes:

- Dutch table index
- English table index
- Dutch sub systems
- English sub systems

- Aliases

Figure 6 shows a sample page. At the top of the page the indexes are listed. Clicking on any one of them will call the corresponding index. The page furthermore lists the columns with their English and Dutch names, any comments (which are currently still mostly in Dutch) and, if any, foreign keys. At the bottom of the page is a table showing all tables that link to this table. Dutch column names, and any table names are hyperlinks calling up another page or showing more detailed information.

Dutch Table Index - English Table Index - Dutch Subsystems - English Subsystems - Aliases

RBH_GEBIEDEN

RBH#FIELDS_

Table alias	GBD
Subsystem	RBH

Dutch Column Name	English Column Name	Comments	Foreign Key
POSTCODE_NR	POSTAL_CODE_NO	woonplaats code	
ID	ID	(TECHNISCH) Systeemgegenereerde identificatie	
CREATE_BRON_ID	CREATION_SOURCE_ID	(TECHNISCH) Referentie naar de bron in ALG_BRONNEN waaruit deze rij ontstaan is.	
CREATE_MOMENT	TIME_CREATED	(TECHNISCH) Datum en tijd van creatie van deze rij.	
CREATE_DOOR	CREATION_BY	(TECHNISCH) Referentie naar de functionaris in ALG_FUNCTIONARISSEN die deze rij heeft ingevoerd.	
LAATSTE_MUTATIE_BRON_ID	LAST_MOD_SOURCE_ID	(TECHNISCH) Referentie naar de bron in ALG_BRONNEN waaruit de laatste mutatie van deze rij afgeleid is.	
LAATSTE_MUTATIE_MOMENT	LAST_MOD_MOMENT	(TECHNISCH) Datum en tijd van de laatste mutatie van deze rij.	
LAATSTE_MUTATIE_DOOR	LAST_MOD_BY	(TECHNISCH) Referentie naar de functionaris in ALG_FUNCTIONARISSEN die deze rij het laatst heeft gemuteerd.	

Foreign Keys

Referring Tables

Dutch Table Name	English Table Name
RBH_ADRESSERINGSGEBIEDEN	RBH#ADDRESS_REGIONS_

OHI Release 10.14.1.0.0; Created on 10-MAR-14

Dutch Table Index - English Table Index - Dutch Subsystems - English Subsystems - Aliases

Figure 6: Example of an html page for a table

3.5. Modification logging

This chapter describes the operation of the modification logging functionality (logging of changes in data) that is available within the OHI BO application.

3.5.1. Why modification logging?

There are various reasons and requirements for equipping an application with modification logging. These can be summarized as follows:

- Traceability
Allows to trace who performed a specific modification and when.
- Modification reports
Provides input for simple overviews of the data modified within a specific period. This can serve a number of purposes.
- Interface support
Save modification details for passing on to other applications in the correct sequence and manner.

Because the application is used by different organizations with varying requirements, modification logging within OHI BO is flexible, enabling fulfillment of the various requirements by means of configuration.

3.5.2. Operation summary

A generic type of modification logging activation has been implemented in the OHI Back Office application, which enables management of each individual table. Consequently, each OHI Back Office functional table has its own table. Furthermore, there is a central log table (ALG_MUTATIE_LOG) in which modifications to specific tables can be logged.

Specific modification logging levels can be activated by means of specific indicators that can be configured per table in the ALG_TABELLEN table.

Modification logging is not activated by default. There is no screen for this activation. Consequently, the relevant columns will have to be updated using (PL/)SQL (with the aid of the ALG_LOGGING_PCK) if required.

There are three types of configuration settings:

1. SOORT_LOGGING

If modification logging (insert, update and delete operations) must be performed in the log table for the table you should set SOORT_LOGGING to: N(ot), B(asic), enabling retrieval of all modifications with the minimum of additional data storage, or U (for Extensive), such that complete records (both new and old values in the event of updates) are placed in the log table.

The latter is used to facilitate easier tracing of who performed which transactions. It is very useful and faster/easier to retrieve what kind of change has been executed but requires more storage space.

2. NIVEAU_LOGGING

If modifications must be logged in the central log table ALG_MUTATIE_LOG set NIVEAU_LOGGING to: N(ot), S(tatement) or for every R(ecord). As a result, the central table can be used to immediately establish the modification types performed, as well as the sequence, and which table(s), without all of the tables having to be examined separately.

3. NUMMERING_LOGGING

If sequence numbers must be distributed set NUMMERING_LOGGING to: N(ot), database T(ransaction) number per set of changes which are contained in one transaction, with a sequence number within transaction per record which is changed, or G(lobal). Global means that an overall modification ID is distributed per statement (insert, update or delete) in addition to the transaction ID and sequence numbers.

A transaction number is used to track a single Oracle transaction (commit) in detail, while the global (statement) numbers are used to determine the overall sequence of statements executed in different transactions.

3.5.3. Default auditing columns

Every 'standard' functional tables has the following set of columns which is also important for modification logging:

Column	Characteristics
ID	<ul style="list-style-type: none">• System generated ID• Uniquely identifies a row within the table• Lends itself for use in generic queries and interfaces.
CREATIE_DOOR	<ul style="list-style-type: none">▪ The user who created the row▪ Refers to ALG_FUNCTIONARISSEN
CREATIE_MOMENT	<ul style="list-style-type: none">▪ The time when the row was created▪ Precision: one second

Column	Characteristics
CREATIE_BRON_ID	<ul style="list-style-type: none"> ▪ The system or interface that was used to create row. ▪ Refers to ALG_BRONNEN ▪ Optional, set by alg_context_pck.set_bron_id
LAATSTE_MUTATIE_DOOR	<ul style="list-style-type: none"> ▪ The user who last updated the row. ▪ Refers to ALG_FUNCTIONARISSEN
LAATSTE_MUTATIE_MOMENT	<ul style="list-style-type: none"> • The time when the row was last updated • Precision: one second
LAATSTE_MUTATIE_BRON_ID	<ul style="list-style-type: none"> ▪ The system or interface that was last used to update the row. ▪ Refers to ALG_BRONNEN ▪ Optional, set by alg_context_pck.set_bron_id

An example of a table (ALG_BRONNEN, see later) is displayed below (the first column and final six columns are the above - mentioned seven columns):

```

ID                NOT NULL NUMBER (14)
CODE              NOT NULL VARCHAR2 (10)
OMS              NOT NULL VARCHAR2 (200)
CREATIE_BRON_ID  NUMBER (14)
CREATIE_MOMENT   DATE
CREATIE_DOOR     NUMBER (14)
LAATSTE_MUTATIE_BRON_ID NUMBER (14)
LAATSTE_MUTATIE_MOMENT DATE
LAATSTE_MUTATIE_DOOR NUMBER (14)

```

When a row is initially created the 'last modification' (LAATSTE_MUTATIE) columns have the same values as the corresponding 'creation' (CREATIE) columns.

Because the references to ALG_BRONNEN and ALG_FUNCTIONARISSEN are nullable, the parent rows may be deleted even if there are still referenced.

Implementing a foreign key constraint at this low level would have too much impact on the performance. It is therefore the responsibility of the customer to keep the ALG_BRONNEN and ALG_FUNCTIONARISSEN tables aligned with the referencing tables.

3.5.4. Configurable modification logging

Because of the various configuration possibilities, the data model features an entity that contains the tables used in the application. This entity also enables entry of various settings per table.

Consequently, modification logging can be configured for each individual table.

However, this only applies to the regular 'functional' data tables (please see the paragraph about availability of modification logging later on). There is no modification logging for technical tables containing system maintained or temporary data, etc.

In general, it can be assumed that the greater the accessibility of the logging data, the higher the required overhead and the greater the storage space required.

Logging must also enable relatively rapid retrieval of specific modifications for disclosure to other parties, for example.

In the light of the above, an implementation method has been chosen that can meet the following configuration requirements which have served as base requirements by means of three settings:

1. Logging?
Is it necessary to log modifications?

2. Log including complete record with new(est) values?

In the event of logging, is it necessary to log only old and modifiable values during an update (after all, the current table contains the new values and fixed values), or is it necessary to log all of the new values in addition to the old values for the purpose of simplicity?

In such cases the records are immediately saved in the logging table with all of the new values in the event of inserts. Furthermore, in the event of updates, all new values are saved in the same record with the modifiable old values.

In the event of updates and deletions, the auditing values from the last update or deletion are placed in the modification record, in addition to the time stamp of the previous modification, regardless of the setting for logging new values.

This can be used to immediately determine the validity of the data (old modification timestamp up to and including the new modification timestamp). In the event of deletions the record is always saved in its entirety if logging is activated to avoid the values being lost.

The unique record ID is always recorded for every logging activity, regardless of the DML operation (insert/update/delete).

This approach enables recording of logging information with minimum overhead if required. This only applies to the previously modifiable values in old columns for which all modifications are traceable.

3. Log unique sequential modification ID in log record?

Is it necessary to record a detailed modification ID in the log record that determines a unique sequence number for all modifications in the application? This number is required additional to a timestamp, as a large number of modifications can be performed within the same second.

Consequently, an ID of this type is necessary to determine the sequence of modifications over various records and/or tables.

Because this number must be generated for each SQL statement (DML) and each record across all tables, the ID generator, although well optimized, may become a 'hot spot', which means it should be used only if necessary.

4. Log statement ID in central overall modification table?

Because the modifications performed within a specific period should easily be retrieved without the need to scan a lot or all of the tables, it is possible to indicate that DML statements that have actually resulted in a modification must be recorded in a central transaction log table (in which an entry is made stating that a record has been inserted in a specific table, for example).

The type of statement (insert, update, delete), executing user account and (optional, see next bullet) involved record are then recorded for the table concerned.

5. Should the modification ID also be logged centrally for all records?

If the modification flow must be produced in greater detail for passing on modifications to another system in exactly the correct sequence, it is possible to indicate that the detailed modification ID must also be recorded centrally for each record, i.e. in the central table. This may result in a greater number of

entries in the central table including a record ID.

Consequently, at this stage only the central table can be used to determine what record types are required from the modification tables in order to reconstruct an outgoing transaction without the various separate log tables having to be scanned (that would be very inefficient to reproduce transactions that span a set of unknown tables; all log tables would need to be scanned for each transaction).

6. Log commit ID in log table?

For certain traceability activities (as well as software problems, for example) it can be desirable to establish which modifications were performed within a specific database transaction and in what sequence.

This can also be desirable in order to make it relatively easy to see by means of which transactions a user produced a specific modification.

This functionality can be activated by means of the generation of a commit ID (an ID for each transaction being committed) in the log table with a sequence number within the commit ID for each modified record (across the various tables).

Please note that commit IDs do not necessarily indicate the actual execution order (for example if multiple short transactions before a long transaction that was started earlier).

7. Should the commit ID also be logged in the central logging table for all records?

The commit ID (commit ID + sequence number per record) can be included in the central table to facilitate easy establishment of the total composition of a logical transaction.

As you see, logging to a log takes place at record level, and not at column level. This is because logging at column level is harder to configure, hardly adds value and has a worse effect on performance.

3.5.5. Example

As an example, let us look at ALG\$BRONNEN, the log table for ALG_BRONNEN:

MUTATIE_OPERATIE	NOT NULL	VARCHAR2 (1)
MUTATIE_ID		NUMBER (14)
COMMIT_ID		NUMBER (14)
SEQ_IN_COMMIT		NUMBER (10)
O\$OMS		VARCHAR2 (200)
O\$LAATSTE_MUTATIE_BRON_ID		NUMBER (14)
O\$LAATSTE_MUTATIE_MOMENT		DATE
O\$LAATSTE_MUTATIE_DOOR		NUMBER (14)
ID		NUMBER (14)
CODE		VARCHAR2 (10)
OMS		VARCHAR2 (200)
CREATIE_BRON_ID		NUMBER (14)
CREATIE_MOMENT		DATE
CREATIE_DOOR		NUMBER (14)
LAATSTE_MUTATIE_BRON_ID		NUMBER (14)
LAATSTE_MUTATIE_MOMENT		DATE
LAATSTE_MUTATIE_DOOR		NUMBER (14)

The modification type (Insert, Update or Delete) contains the type of operation that led to the creation of the log record.

The modification ID and the commit ID are given values if so configured.

The columns in the table that can be modified by the user or by means of the code are included twice: once as old columns (the name is based on the column name in the original table, prefixed with O\$) and again as 'new' columns (same column names as in the original table).

Columns that cannot be modified are included as 'new' columns, for example the ID column.

When basic logging is activated for the sole reason of traceability, a record containing all modifiable columns with the values prior to the record modification (regardless of whether the modifiable column concerned has also been modified) is saved in the event of a modification to a modifiable column.

This makes it possible to reconstruct any past situation in combination with the current record (uniquely identified by means of its ID).

The name of a log table can be derived from the original table (see 'Tables and views'). For example ALG\$BRONNEN is the log table for ALG_BRONNEN. Likewise, ALG#SOURCES\$ is the 1:1 view on ALG\$BRONNEN.

When the composition of a transaction, and the sequence in which modifications are performed across the various tables, needs to be established without large numbers of scans of all log tables, it is possible to use additional logging of references to the log tables in the central overall log table.

The central OHI BO log table ALG_MUTATIE_LOG is structured as follows:

MUTATIE_OPERATIE	NOT NULL VARCHAR2 (1)
TAB_ID	NOT NULL NUMBER (14)
RECORD_ID	NUMBER (14)
COMMIT_ID	NUMBER (14)
SEQ_IN_COMMIT	NUMBER (14)
MUTATIE_ID	NUMBER (14)
MUTATIE_DOOR	NUMBER (14)
MUTATIE_MOMENT	DATE

This table contains a reference to the table containing the table names, which facilitates easy tracing of the table in which the centrally logged modification was performed. The column containing the Record ID refers to the unique number per table issued to each record.



Note: The COMMIT_ID is not the same as the SYSTEM COMMIT NUMBER (SCN).



Note: The ID column contains the unique ID of the row in the original table. Note that since many modifications to the same row can be logged, this ID is not unique!



Note: There are no application defined indexes present on log tables except for when classic purging functionality has been present on such a table, in which case the ID column is indexed.

3.5.6. Activation

The settings for modification logging are maintained in the ALG_TABELLEN table.

The ALG_LOGGING_PCK package can be used to control the activation and deactivation of logging/journaling of functional tables. Alternatively, these settings can be set by manually updating the ALG_TABELLEN table.

The ALG_LOGGING_PCK offers the following routines that require a table name as parameter value to implement the functionality:

- DISABLE_LOGGING - disables logging to the central table but leaves logging/journaling records to the log table as specified earlier
- DISABLE_JOURNALLING - completely disables any kind of logging for the table, whether it is the central table or the table-specific log table
- ENABLE_JOURNALLING_BASIC - enable logging to the log table with the least amount of details (requires less space than full journaling)
- ENABLE_JOURNALLING_FULL - enable logging to the log table while storing the complete record and old and new values
- ENABLE_LOGGING_STMT - enables logging of statement executions on a table to the central log table with transaction id and modification id sequence numbers being assigned
- ENABLE_LOGGING_ROW - enable logging of each involved row that is changed by statement execution to the central log table with transaction id and modification id sequence numbers being assigned
- PURGE_LOG_TABLE - purge the data from the central log table and the specific log table until the given date (see separate paragraph for more details)



Note: When specifying a table name, be sure to use the original table name, for example ALG_BRONNEN instead of ALG#SOURCES_

If you want to use other combinations, for example logging of statements to the central log table but without a global modification id being assigned, you should update the ALG_TABELLEN table directly.

The best way to determine what is suitable for your situation is to just start with enabling logging on some tables, enforce some statements that generate logging and evaluate the results in the log tables and central log table.

Note that changes to the modification logging may not immediately effective:

- Of course you need to commit your changes in order to make them visible for other sessions.
- The settings for modification logging are cached at the session level. In order to force a reload, sessions must be restarted.

The above-mentioned configuration options per table are available for all regular functional tables that are directly available for executing inserts, update or deletes. These tables implement business rule validation through a series of triggers and constraint definitions. The business rule validation mechanism implements also this modification logging functionality.

This means that other tables, which cannot directly be modified, do not offer this functionality. Such tables are typically maintained by the application. These application-maintained (or code-maintained) tables are often referred as 'technical' tables in comparison with the user maintainable tables as 'functional' tables. The technical tables can be recognized as having a '#' sign on the fourth position of their name.

The standardized way of granting insert, update, delete and select privileges makes sure only the regular 'functional' tables can be changed.

3.5.7. Impact of release upgrades

When a new OHI release is installed an important requirement is to minimize the required installation time in order to have a minimal downtime of the application.

For that reason changes in the data structure or in the data as result of an OHI release installation (a single patch, a patch set or a major release) are often optimized. These optimizations may result in the following consequences for the logging data (although they often do not apply):

- Changes due to a scripted update are not logged
- The table structure is changed and the current contents are converted in the table (by adding for example a default value for a new column) while the logging table is not updated
- A table may be newly (re)created where the existing data is converted to the new structure; these changes will not be reflected in the logging table
- The log table might need to be recreated or dropped; in which situation the old contents are dropped

These consequences do not apply to the central logging table.

Although these consequences might look severe, the heavier the consequence the less often it occurs. And it must be considered that these are completely scripted repeatable operations. For a potential impact of these types of unlogged changes we advise to determine the 'delta' between the situations before and after a release installation to judge whether such modifications are relevant for the derived functionality. Only when there are consequences these must be implemented on the derived environment.

In the rare situation when traceability is required and a log table is cleared we advise to export the table contents before implementing the release installation on the production environment. The saved contents can be used for a custom conversion or later retrieval when necessary.

3.5.8. Purging logged data

DML operations are not granted to the individual users on the logging tables.

To purge data for a given log table the procedure PURGE_LOG_TABLE is available in the ALG_LOGGING_PCK. The procedure expects a table name and a date in the past upon which changes should be purged (so including that date).

The function will purge all the data for the given (functional) tablename from the specific log table and the central log table. The data will be purged until the given date. However, to prevent recent and potential still relevant data is purged this date should be at least one year in the past. As purging is done using full table scans even purging one log record may still take considerable time, depending on the log data which is present.

3.6. Tracing

The application code of OHI Back Office has been instrumented to trace the PL/SQL code as it is executed. Both the CAPI packages, batch packages and underlying packages of the OHI Back Office Business Services have been instrumented.

By turning on tracing you get a detailed overview to help you analyze the program flow. This is useful if you want to understand a problem or need to send extra information to customer support.

Another use may be to instrument your custom code with trace calls.

See 'Appendix C - Tracing' for details.

4. Flex Fields

As a standard solution for healthcare payers, the OHI Back Office data model cannot be altered. However, the OHI Back Office data model can be extended with so-called flex fields to register additional customer-specific data, for example to record when members had their last health check or to register parameters for interfacing with providers.

These flex fields have the following benefits:

- The core data model stays unchanged, which means that OHI Back Office can support the application as sold to the customer.
- The customer does not need to create custom tables, which must be kept in sync with the OHI Back Office database.

Historically, flex fields were used for claims line processing and to add extra data to policies and relation data. Soon after, flex field implementations were created for other tables as well.

Flex field support will be extended in future releases.

4.1. Concepts

The current generic flex field implementation is introduced in release 10.13.1.0. These flex fields can be accessed in the GUI application through the key combination CTRL-SHIFT-F2.

Current support for flex fields:

	Historic	Generic
Included in XML output	X	-
Included in CSV output	-	-
Access through GUI	X	X
Query through GUI	-	-
Create or drop flex fields through GUI	Y	Y
TAPI support for on-cascade delete of flex fields	Y	Y
PLSQL supplied packages to support queries on flex fields and flex field values	-	-
PLSQL supplied packages to support to perform DML on flex fields and flex field values	-	-
PLSQL supplied packages to support to create or drop flex fields and flex field values	-	-
Access through (SVL) web services	X	X
Allow dynamic SQL to validate flex fields	-	-

In time both flex field implementations will be consolidated into a single solution.

4.1.1. Related

The claims processing code uses a specific flex field mechanism which is used to add context-specific data to the claims calculations. In time this mechanism may be migrated to the current generic flex field implementation.

4.2. Flex Field Characteristics

These characteristics apply to the current generic implementation.

4.2.1. Flex field Types

In practice there are two flex field types:

- attributes
A number, character string or date value, which may be restricted to a range or set of values.
- key references
The value contains a reference to an existing row in the OHI Back Office database.
- care role
Key reference to a care provider.
- relation role
Key reference to a relation

Both 'care role' and 'relation role' are precursors of the new key reference implementation and may be migrated to a key reference in due course.

4.2.2. Filters

There are two types of filters for flex fields:

- Flex field scope
Indicates in which context a flex field value can be used (core processing).
- Flex field group
Evaluates whether a group of flex fields can be set at all.

4.2.3. Flex Field Scope

Indicates for which context a flex field value can be used.

Possible contexts are:

- Benefit extent
- Benefit threshold
- Premium amount
- Registered
- Yearly deductible amount

A flex field may have multiple scope records.

Note also that flex field scope records are temporal (time-valid).

4.2.4. Flex Field Groups

The flex field group definition can be used to restrict the use of a group of flex fields for a given row in the master table.

In that case, a dynamic PL/SQL unit, configured for this group definition, is evaluated to decide whether the group of flex fields can be set.

This evaluation takes place:

- When selecting from the REF_EWE_VW view (for example in the flex field maintenance screen (ZRG7205F))
- When inserting flex field values for the given master table (for example through PL/SQL or web services).

Note:

- This is a much more restrictive filter than the flex field scope (which allows the flex field to be set and allows its use in a given context).
- The dynamic PL/SQL code can access the columns of the master row through `tbl_rec.<column_name>`.
- Since the evaluation of the PL/SQL definition is done dynamically, a later evaluation may be more restrictive. In that case previously defined flex fields will still apply.

The flex field group definition is also used to display the flex fields in a group in ZRG7205F.

4.3. Flex Field definition

Screens:

- ZRG7027F - Maintaining Entity Flex Fields
- ZRG7019F - Flex Field
- ZRG7206F - Flex field Group

Typically you would start with ZRG7027F:

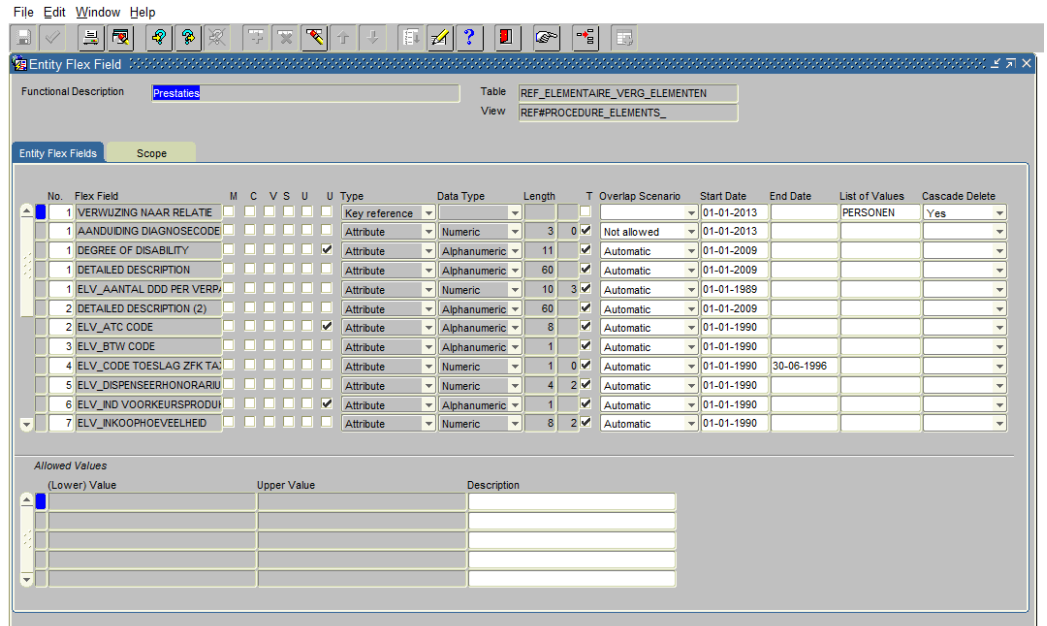


Figure 7: Maintain Entity Flex Fields

4.3.1. Candidate tables for flex fields

Run the following query which will list all tables for which flex fields can be defined by their English view name:

```
select english_name, name from alg#tables_
where flex_fields_ind = 'J' order by 1;
```

4.3.2. Allowed values

You may specify the criteria for the value of a flex field, for example using ranges (lower + upper boundary) or allowed values (low value + description).

If you do not specify any allowed values, any valid value for the given data type is acceptable.

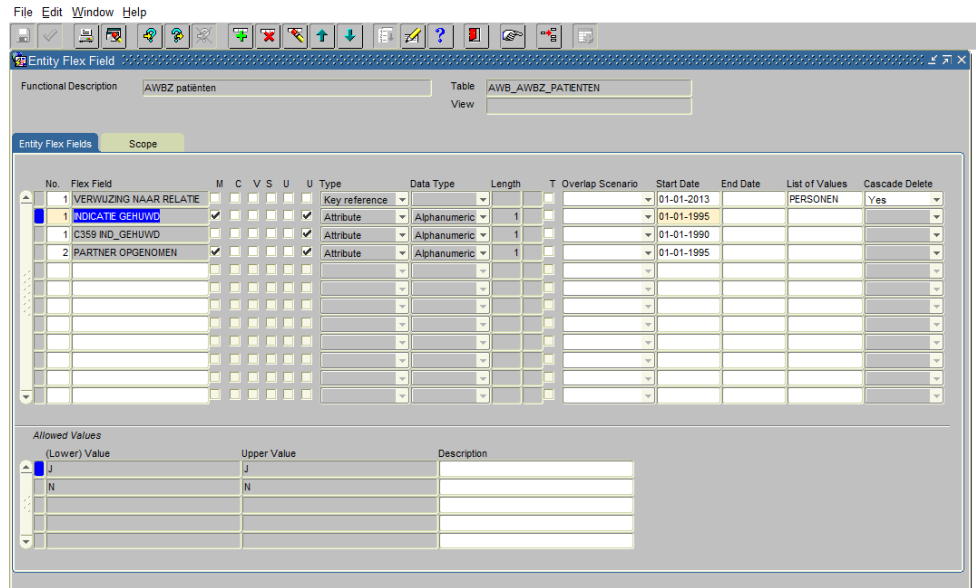


Figure 8: Specify allowed values

4.3.3. Time-valid flex fields

If you define a flex field to be time-valid, you should also specify the overlap scenario (automatic, permitted, non-permitted).

Do not confuse the validity of the flex field value with the validity of the flex field definition.

4.3.4. Multi-value flex fields

A multi-value flex field is used to hold a set (or array) of values valid at the same point in time.

For example, for registering multiple parameters when registering details to connect with an interface.

4.3.5. Key Reference

The flex field value refers to a record in another table.

Use 'List of Values' to select a LOV query to present the runtime user with a LOV screen from which to select the FK reference.

Think about setting 'cascade delete':

- N : the deletion process for the parent of the key reference will only succeed if there are no flex field values referencing to it.
- Y : the deletion process for the parent will remove the flex field values referencing to it.

You may want the context menu to start the List Of Values Definitions Screen (SYS1152F):

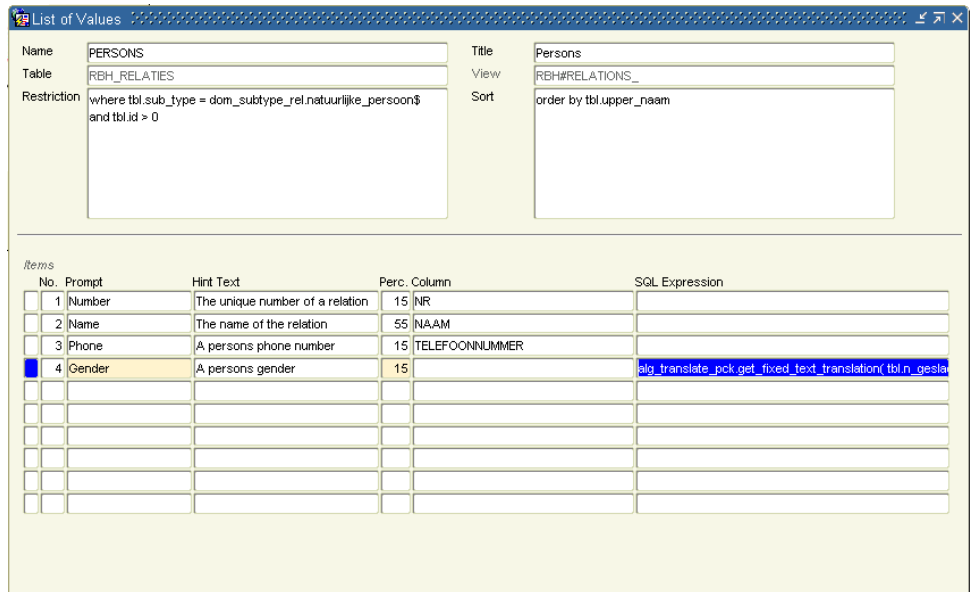


Figure 9: Set up a list of values

4.4. Flex Field maintenance

Flex fields for the current generic implementation are maintained with ZRG7205F which is invoked through the CTRL-SHIFT-F2 key combination:

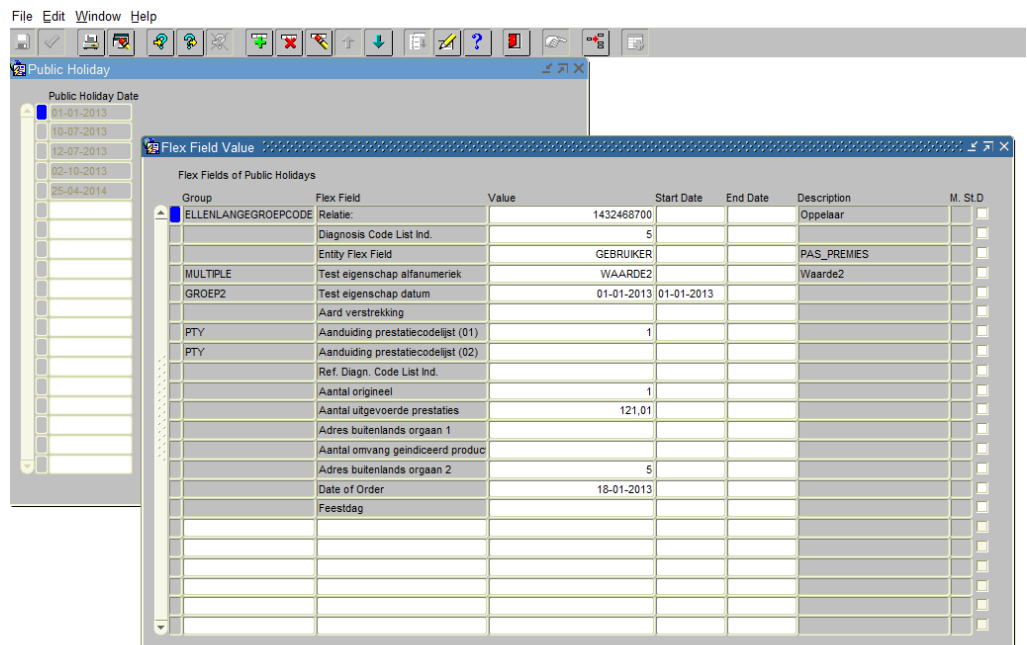


Figure 10: Entering values for Entity Flex Fields

4.5. Indexed access of Flex Field values

Normally flex field values will be queried within scope of their 'parent' record. For example, if the length (in centimeters) of a specific person is recorded using a flex field, this property will be accessed using the index that contains the reference to the person record itself. When a large set of person records is queried for a specific value of that flex field (regardless of the person), this is not efficiently supported by default (a full table scan is required as indexed access is not possible by default). However, a facility is present in OHI Back Office that enables indexed querying of flex field values stored in table REF_EIGENSCHAP_WAARDEN. There is support for entity

flex fields of data type CHAR and NUMBER. For making an entity flex fields' values 'Searchable' see the online help on this subject.

This facility should be used with great care. In most implementations of OHI Back Office the flexfield value table is one of the largest tables in terms of number of records. If too many flex fields are made 'Searchable' this will have a negative impact on performance and storage requirements. The index may become rather large to support this access path. Time is also involved in updating the index when new value records are added for these flex fields.

Because the indexes used are function based, the function used to populate the index should also be called when querying for specific flex field values.

For indexed access of the flex field definition (column EGE_ID): `ref_ewe_index_ege(ewe.ewe_id, ewe.tech_ind_index) = <ID of the flex field definition>`

For indexed access of flex field values of type CHAR (column WAARDE_CHAR): `ref_ewe_index_char(ewe.waarde_char, ewe.tech_ind_index) = <VALUE of the flex field>`

For indexed access of flex field values of type NUMBER (column WAARDE_NUMBER): `ref_ewe_index_number(ewe.waarde_number, ewe.tech_ind_index) = <VALUE of the flex field>`

It is advised to use the condition on the flex field definition and the flex field value simultaneously. The optimizer can determine the best access path using these function calls.

Example query:

```
select record_id
from   ref_eigenschap_waarden ewe
where  ref_ewe_index_ege(ewe.ewe_id, ewe.tech_ind_index) = 1000000000248 --
l_ewe_id
and    ref_ewe_index_number(ewe.waarde_char, ewe.tech_ind_index) < 190 --
l_length
;
```

5. Dynamic PL/SQL

OHI Back Office's support for dynamic PL/SQL is a powerful mechanism which allows the customer to add customer-specific code to the core OHI Back Office product.

OHI Back Office has defined several 'hooks' in OHI Back Office where custom code can be added e.g. for:

- Data entry validation
- Address entry and formatting
- Trigger conditions for business event framework
- Core processes such as claims processing, policy collection and payment processing.

Apart from adding a high level of flexibility to the customer our support of dynamic PL/SQL helps to keep the amount of code in the standard product under control.

Note that all dynamic PL/SQL code is executed under the OHI_DPS_USER account (more info in the 'Authorization' paragraph of the 'Open Database' chapter).

If you want to call other custom developed packages from dynamic SQL you should:

- Grant execution privileges of your custom developed packages to OHI_DPS_USER
- Create public synonyms for your custom developed packages or prefix calls to your custom developed package with the package owner.

5.1. Hooks for PL/SQL code

OHI Back Office has defined several 'hooks' in OHI Back Office where custom code can be added:

- Column value validation
- Commission calculation
- Claims Processing
- Policy Creation
- Policy Collection
- Population Register Check
- XML Processing
- Address Entry
- Address Display
- Trigger conditions for the business event framework
- Payment Processing
- ..

It is expected that this list will grow in time.

5.2. Dynamic PLSQL Definition

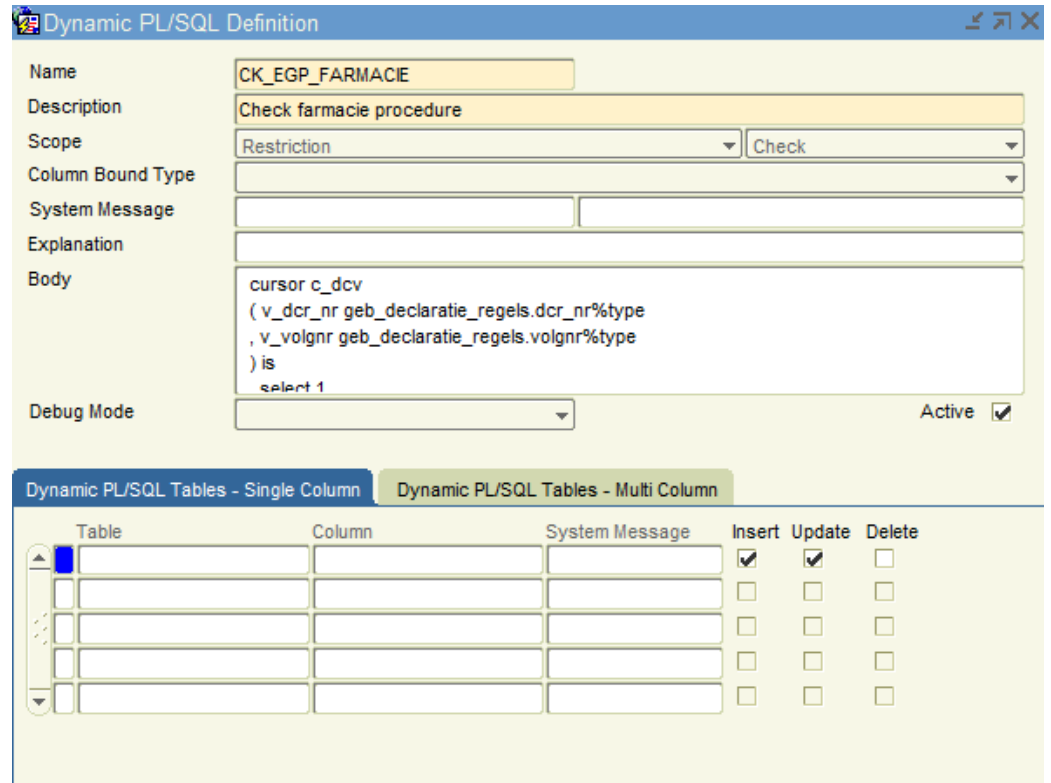
The process of adding dynamic PL/SQL boils down to this:

- Select the hook or 'scope' where you want to insert custom code.

- Create the custom code in the format required by the 'subtype' associated with the scope.
- Extend the application to specific tables or columns if you are adding a column validation.
- After testing, revise the debug level.

5.2.1. Setup

The PLSQL definition is maintained in the screen "Dynamic PL/SQL Definition" (SYS1139F)



The screenshot shows the 'Dynamic PL/SQL Definition' window. The 'Name' field contains 'CK_EGP_FARMACIE'. The 'Description' field contains 'Check farmacie procedure'. The 'Scope' dropdown is set to 'Restriction' and the 'Check' dropdown is set to 'Check'. The 'Column Bound Type' dropdown is empty. The 'System Message' and 'Explanation' fields are empty. The 'Body' field contains the following PL/SQL code:

```

cursor c_dcv
( v_dcr_nr geb_declaratie_regels.dcr_nr%type
, v_volgnr geb_declaratie_regels.volgnr%type
) is
select 1

```

The 'Debug Mode' dropdown is set to 'Active'. Below the main form are two tabs: 'Dynamic PL/SQL Tables - Single Column' and 'Dynamic PL/SQL Tables - Multi Column'. The 'Single Column' tab is active, showing a table with the following columns: Table, Column, System Message, Insert, Update, and Delete. The table has five rows, with the first row having 'Insert' and 'Update' checked.

Table	Column	System Message	Insert	Update	Delete
			<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Figure 11: Dynamic PL/SQL definition

In addition to the online help for the SYS1139F screen, note:

- When making changes, ensure that 'Active' is unchecked.
- The upper part of the screen is used to define the custom PL/SQL code and in which context (scope/subtype) it is used.
- Make sure that the 'name' and 'explanation' clarify the purpose of your custom code.
- The 'body' attribute is initialized with sample PL/SQL code when you select the scope for the first time.
- The attribute 'column bound type' as well as the 'tables' and 'columns' tabs in the bottom part of the screen are meaningful only if you have selected the 'column bound' scope, ie. if you are going to validate database column values.

5.2.2. Scope and subtype

Dynamic PL/SQL is executed at runtime by the OHI Back Office code.

The 'scope' defines WHEN the PL/SQL code is executed.

The 'subtype' defines HOW the PL/SQL code should look like, i.e. the input and output parameters.

The scope and the sub type of a scope together constitute a Dynamic PL/SQL Usage Type or 'usage type' for short.

You cannot create new usage types, but you can query the predefined usage types in the SYS1138F screen (Dynamic PL/SQL Usage Type).

The SYS1138F screen is particularly useful to find the possible bind variables and output types.

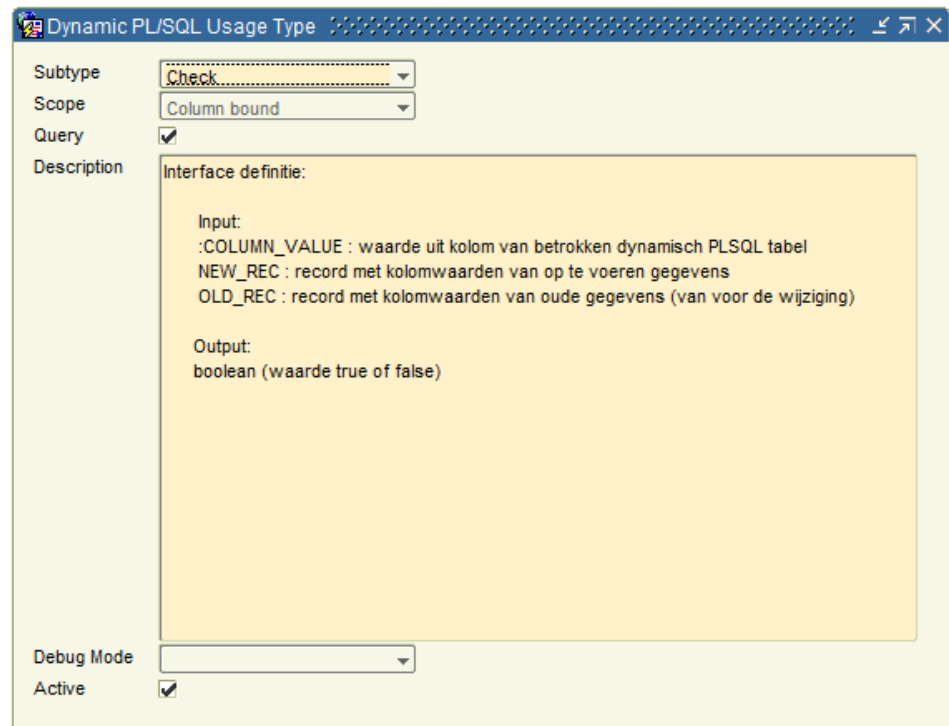


Figure 12: Dynamic PL/SQL Usage Type



Note: bind variables

Some pieces of dynamic PL/SQL are defined to use bind variables. If you want to omit a bind variable from your code, OHI will fail to bind a value to your code at runtime and abort with an error. This means that the bind variable must be in the code, even if it is just in comments.

Please find a more detailed list with hooks in 'Appendix F - Dynamic PL/SQL Types'.

5.2.3. Throwing or adding messages

It is possible and allowed to use the message handling routines from the package SYS_MESSAGE_HANDLING_PCK, like GIVE_ERROR or GIVE_INFORMATION.

The message handling is not interactive, which means a message can interrupt the code by raising an exception or does not interrupt and does proceed with processing the code after the call of the message routine.

When you throw an error this raises an exception (unless you specify an exception should not be raised). In other situations it depends on the context what happens to the message: in a batch context the message is stored in the database but in an

interactive situation it is added to the stack and only shown when at a later stage during the same process call an error is thrown.

The advantage of using this explicit calls of message routines is that you can use different messages for different values and/or pass values for substitution parameters in a message. So this offers more freedom than specifying only a message code for a 'check' like discussed in the paragraphs that follow.

An example (where the substitution parameter should be in the actual message, the default text is only meant as documentation and for when the message is not present in the message definition table to have a fallback):

```
sys_message_handling_pck.give_error
( pi_msg_code      => 'ALG2222'
, pi_msg_default_text =>
  'The provided value should be positive.'
, pi_msg_parmvalue_tab =>
  sys_message_handling_pck.msg_parmvalue_tabtype
  ( sys_message_handling_pck.msg_parmvalue_rec
    ( pi_sequence_nr => 1
    , pi_name        => 'Value'
    , pi_value_char  => l_record.amount
    )
  )
, pi_raise_exception => true
);
```

5.3. Column bound checks

Column-bound checks are executed when inserting, updating or updating records.

Many of these column-bound checks have been implemented by OHI Back Office, so you will only need to create additional checks.

Examples of column-bound checks:

- Syntax check of dynamic PL/SQL code
- Check whether an IBAN number is valid
- Validate that the end date of a record must be at least 14 days after the start date.
- ..

A column-bound check evaluates to 'true' or 'false'. There is no 'somewhat true'. If all column values for a given row are evaluated to 'true', the DML operation (insert, update, delete) will continue.

If one or more checks evaluate to 'false' the DML operation will abort and the error message defined for the failed column check will be shown.

Note that a single column-bound check can be applied to more than one table, provided that each column is of the same type and name as the column in your custom PL/SQL code.

5.3.1. Single column checks

A single column bound check uses one column to check. A typical example of a single column bound check is the validation of a format mask for the given column.

Let's use the following example in which a telephone number must always begin with a "+" sign. The telephone number is stored with a relation, so it is a check on a

column in the relation screen. If the telephone number does not start with a “+” an error will be presented to the user.

5.3.1.1. Error message definition

Custom error messages must be defined using the screen “System Messages” (Menu option: System / Management / General / System Messages”), as shown in Figure x.

We will not discuss message creation here in depth, but will only mention a few points:

- Custom messages can have any code, but it is good practice to start them with the code “SVS”. Although not enforced for messages most of the custom code(s) in OHI must start with SVS.
- Custom messages must belong to the subsystem “External batches”.

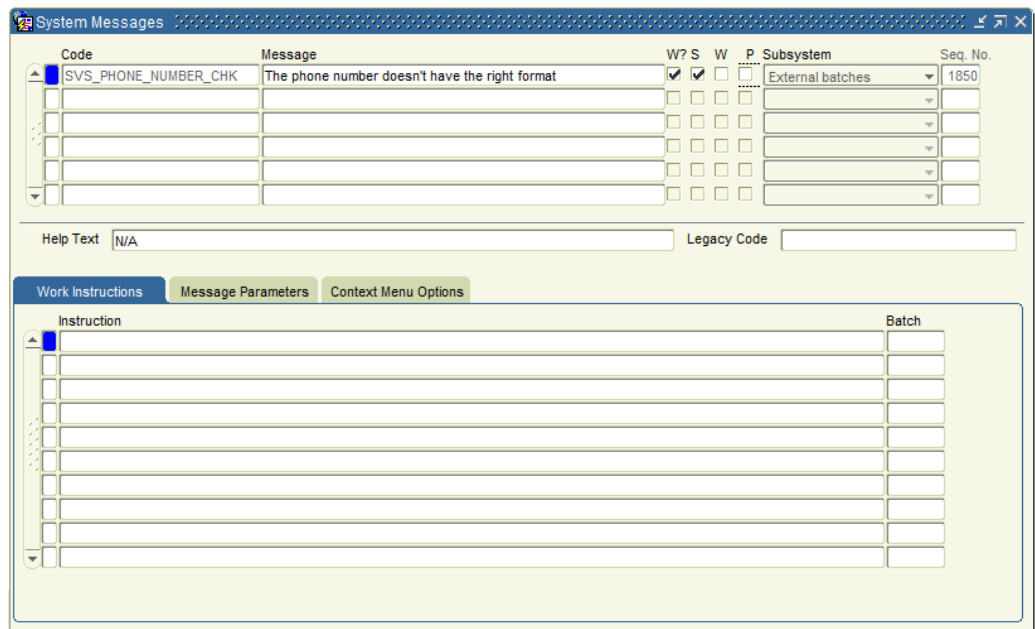


Figure 13: shows the definition of an error message for our example.

5.3.1.2. PL/SQL definition

Next step is creating the Dynamic PL/SQL definition. This is done in four steps:

1. Create a Dynamic PL/SQL definition. This means we need to choose a Name (or Code), a Description, the Scope (“column bound” in this case), a Column Bound Type (“Single-Column” in this case) and the System Message that we defined in the previous paragraph.

Note that the Active check box must be unchecked. The reason for this is that the record must be saved before we can add Tables and Columns in the lower section. But saving the record with the Active flag checked will cause OHI to check the syntax of the PL/SQL code which needs a reference to the tables and columns, which in turn have not been added yet. (See Figure 14).

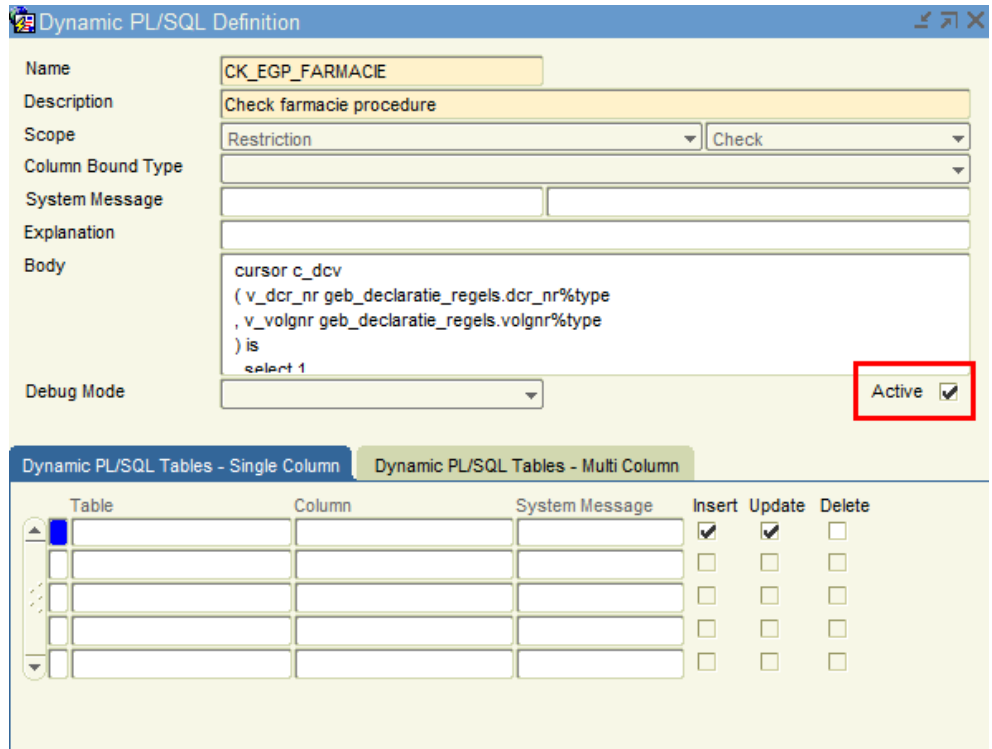


Figure 14: step of creation of a Column Bound check.

2. Add the tables and columns in the lower block, in the “single column” tab. In our case this means that we have to provide the Relations table and two columns, because there are two telephone numbers that can be stored in the Relations table.

Currently only the Dutch table and column names are accepted as input. In our example we only need to check the value upon insertion or update of the telephone number, so only the Insert and Update check boxes are checked. See Figure 15.

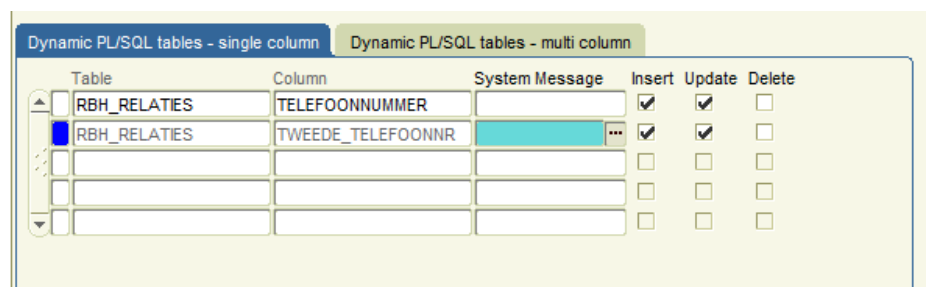


Figure 15: Adding table and columns to PL/SQL definition.

3. Create the actual PL/SQL. See below for the code for our example.

```

-- declaration section
-- bind variable: :COLUMN_VALUE
  l_retval boolean := true;
begin
  -- body section

```

```

if :COLUMN_VALUE is not null
then
  l_retval := substr(:COLUMN_VALUE,1,1) = '+';
end if;
return l_retval;
end;

```

Note:

- a. The code can use three variables:

The bind variable - :COLUMN_VALUE, which contains the new value of the column.

The variables OLD_REC and NEW_REC which contain the old and new record for the same row. When inserting, the OLD_REC contains only null values. When deleting, NEW_REC is empty. Only in case of an update do both OLD_REC and NEW_REC have values.

- b. The code must return a boolean value. If the new value passes the test TRUE must be returned, in which case the new value will be committed to the database (if no other error occurs!). If FALSE is returned the message provided in the Dynamic PL/SQL definition will be displayed to the user, and the new value will not be committed to the database.
 - c. :COLUMN_VALUE is a read-only value. You cannot change this to fix a problem.
 - d. Be aware of NULL values. When activated the column bound check is performed always regardless whether the column has a value or not, or whether it changed or not. Hence the check on the NULL value in the code in Figure x.
4. Activate the Dynamic PL/SQL (Figure x).

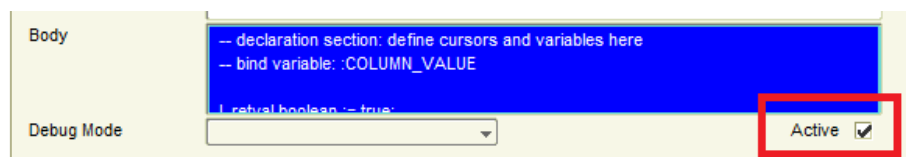


Figure 16: Activating the dynamic PL/SQL definition.

5.3.2. Multi column checks

A multi column bound check uses more than one column for its check, but is in all regards similar to a single column bound check.

Let's use the following example: an organization must have a web site. An organization is stored in the same Relations table as a regular person. There is, however, a column indicating whether a relation is an organization or a person. This column needs to be checked together with the field for the URL.

5.3.2.1. Message definition

Just as for the single column check we need an appropriate error message. Figure 17 shows our message. Again we have the code of the message start with SVS.

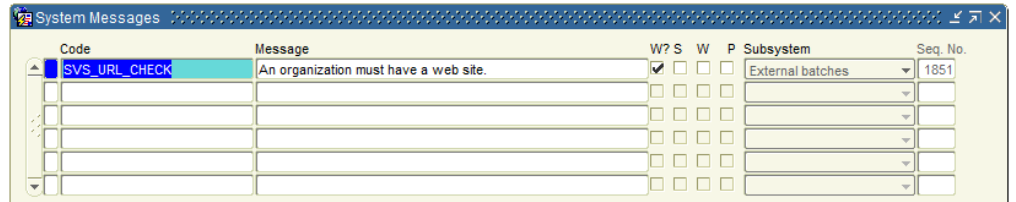


Figure 17: Error message for multi column bound check example.

5.3.2.2. PL/SQL Definition

The PL/SQL definition differs in only one field with a single column bound check. The Column Bound Type must be set to Multi-Column. See Figure 18.

Again we need to make sure that the check is not activated yet, in order to prevent OHI from checking the PL/SQL code syntax.

The message we provide is the one from the previous paragraph.

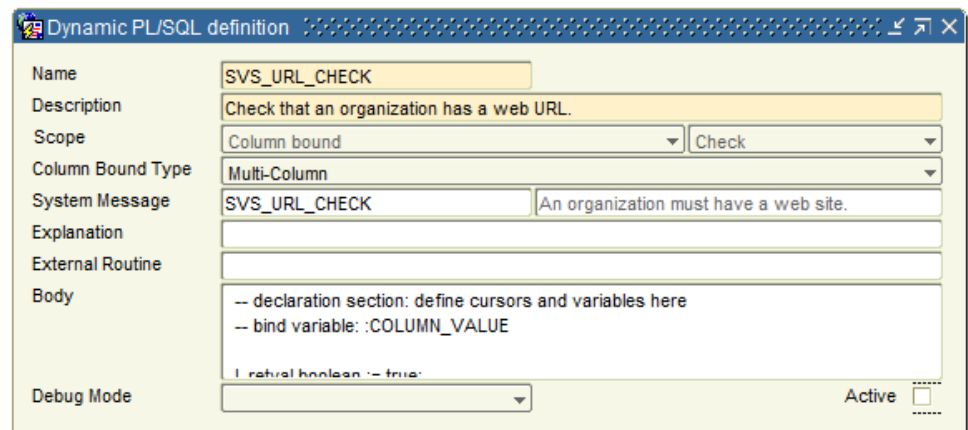


Figure 18: Dynamic PL/SQL definition for Multi Column Bound Check.

Next is the selection of tables and columns, on the second tab in the bottom of the screen (Figure 19). The table name must be provided in Dutch, as must the column names. The table is RBH_RELATIES and the columns we are interested in are SUB_TYPE and URL.

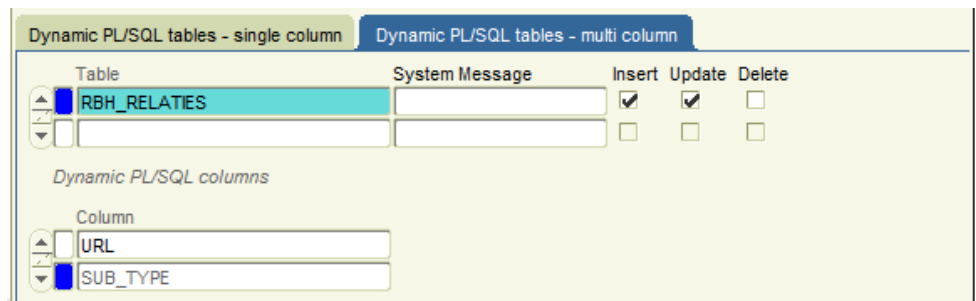


Figure 19: Table and column definitions for multi column bound checks.

Now for the PL/SQL code. This is shown in Figure x. Note that it is not needed to check new_rec.sub_type for a null value as this field is mandatory in OHI. If the sub_type is equal to 'O', which means the relation is an organization, the return value is determined by checking the URL column for null. If it is not null a url has been provided for the organization and true will be returned, otherwise false.

Note that the bind variable :COLUMN_VALUE cannot be used as OHI does not know which of the specified column values should be used. Only the variables OLD_REC and NEW_REC can be used. But even though the bind variable cannot be

used it still must be mentioned in the code. So put it in the comments (which, by the way, is done by default).

Example code

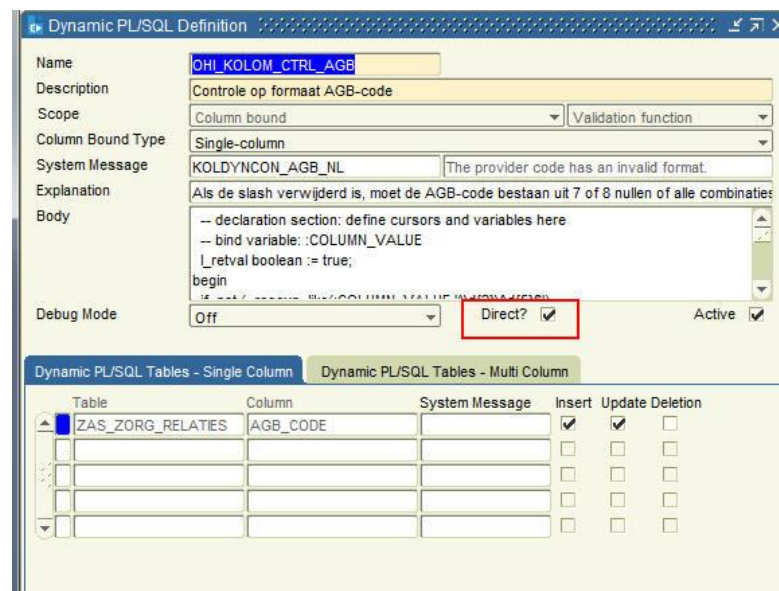
```
-- declaration section
-- bind variable: :COLUMN_VALUE
  l_retval boolean := true;
begin
  -- body section
  if new_rec.sub_type = 'O'
  then
    l_retval := new_rec.url is not null;
  end if;
  return l_retval;
end;
```

After saving the dynamic PL/SQL definition can be activated in the same way as for single column bound checks, see Figure 16.

5.3.3. Direct or postponed checks

In some cases a check needs additional information from other records which can be mutated in the same transaction. This could be as simple as a check on the minimum or maximum number of allowed details for a certain master.

A direct check, or statement level check, immediately checks after the creation, update or deletion of a record if this record complies with the check. Deactivating the 'Direct?' checkbox will postpone the check until the whole logical transaction is ended. This allows you to check the situation that all data is changed from the master as well as the detail records and complies to the check.



5.3.4. Activating and deactivating column bound checks

When unexpected code errors occur on screen, these may be related to the use of column bound checks, both single and multi column. To quickly determine whether such a column bound check is responsible for the error there are two ways:

1. Deactivate the dynamic PL/SQL definition. This will prevent the execution of the check. The disadvantage of this approach is that the check will be disabled for all tables and columns it is applied to.

- Use the screen "Tables" (Menu option: System / Management / General / Tables, tab: Column bound checks). This enables the developer to selectively disable the check for a single column bound check. Furthermore, the table provides an overview over all checks that were put in place on a table. Figure 20 shows the list of custom checks that were defined for the relations table.

Name	Description	Column	Type	Active	Insert	Update	Delete	Active
SVS_PHONE_NUMBER_C	Check telephone number	TELEFOONNUMMER	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
SVS_PHONE_NUMBER_C	Check telephone number	TWEEDE_TELEFOONNR	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
SVS_URL_CHECK	Check that an organization has a web URL.		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Figure 20: Listing of all column bound checks for table RBH_RELATIES.

5.4. Writing custom code

It is advisable to create a PL/SQL package in your custom schema and limit your Dynamic PL/SQL code to packaged function calls.

Example:

Suppose you have currently your Dynamic PL/SQL body like this:

```
-- :COLUMN VALUE
  l_retval boolean := true;
begin
  if api_rbh_util_pck.get_rns_vrij_tekstveld(pi_rsn_id => new_rec.rsn_id) =
  'IBAN CONTROLE'
  then
    l_retval := api_rbh_util_pck.check_iban(pi_rekeninggegevens =>
new_rec.rekeninggegevens);
  end if;

  return l_retval;
end;
```

You can create your own validation function and limit the dynamic PL/SQL code to:

```
-- :COLUMN_VALUE - reference to bind variable
begin
  return custom.validate_pck.check_iban(pi_erk => new_rec);
end;
```

The advantages:

- You can keep all your custom validations in one place.
- It is easier to move your code from one environment to another.
- You can use your favorite tools to develop and test your code.

6. Business Event Framework

With the Business Event Framework you can define customer-specific events and event handlers.

These events can be time-based, triggered by a change or created by your custom-designed detection mechanism.

The handlers can be run near real-time or in batch processing mode and are implemented through custom PL/SQL code. Note that the custom PL/SQL is executed under the OHI_DPS_USER account.

6.1. Overview

Specific hooks are required in the OHI Back Office application for customers to develop custom event handling using the OHI Back Office database. The Business Event Framework can be used to signal specific events in the OHI Back Office application. These events can arise from creating or modifying data or by the passing of time. The framework is also used to define how an event should be handled.

Since the majority of custom development for OHI Back Office implementations is PL/SQL based, the framework is implemented in PL/SQL.

The Business Event Framework provides two options both for signaling and responding. These options can be combined for each business event to create the most suitable environment for handling the event.

An example of how the Business Event Framework can be used is when a member supplies the health care payer with their change of address after relocating. The health care payer has an integrated customer relationship management (CRM) system and uses the change of address event to automatically trigger an update to the CRM database.

6.2. Signaling Events

The Business Event Framework offers two options for signaling events and both are described in this section.

6.2.1. Detected Events

Detected events are events that are signaled by querying the data in one or more tables. A decision to register the event is based on the results of the query. The event is registered based on the data that was found at the moment the data was queried. This moment can be controlled by scheduling Process Business Events (SYS5001S) batch (see Starting Business Event).

For example, a relation record is updated at 08.30, 11.15 and 14.50 hours. When the batch is scheduled to run at 15.00, the data from the last modification (14.50) will be evaluated. The data for the record as at 09.00 or 12.00 cannot be signaled by a detected event.

Detected events are best used in situations where the intermediate modifications are not important or where the passing of time is the trigger for the event.

6.2.2. Triggered Events

Triggered events are signaled the moment they occur. Using database triggers an event is evaluated and registered. Unlike the detected events, intermediate changes can be signaled. In the relation record example, which is updated at 08.30, 11.15 and 14.50, a triggered event can be registered for all three updates.

Events can be signaled separately based how the data is modified, for example insert, update or delete.

Triggered events are used to signal data modifications immediately.

6.3. Responding to Events

The Business Event Framework offers two options for responding to events and both are described in this section.

6.3.1. Batch Response

To process signaled events in a batch the signaled events must be stored in an OHI Back Office table. The moment the event is signaled, either through a detected event or a triggered event, the event is saved to table ALG#EVENTS. The Process Business Events (SYS5001S) batch handles the events. The batch can be scheduled to run at the correct intervals (see Starting Business Event).



Note: No duplicate events

Duplicate events will not be stored when saving events to a table. In case a relation or policy is signaled multiple times for the same event, the table will hold only one occurrence of the event. If the same data manipulation type is performed twice for an event on a table and record, only the first will result in an event.

After successful processing the event, the same event could be detected again.

6.3.2. Near Real Time

When an event should be signaled the moment it occurs, events can be stored to a queue. An OHI Back Office background process is continuously listening to the queue. Events are taken from the queue and processed immediately.



Note: Queued events are processed by a separate process (with its own database session). This may result in locking issues if the handler and the originating process both want to update the same record.

6.4. Combining Signaling and Response Types

Four definitions result from the two types of events together with two storage options. This section describes the situations where each definition can be used.

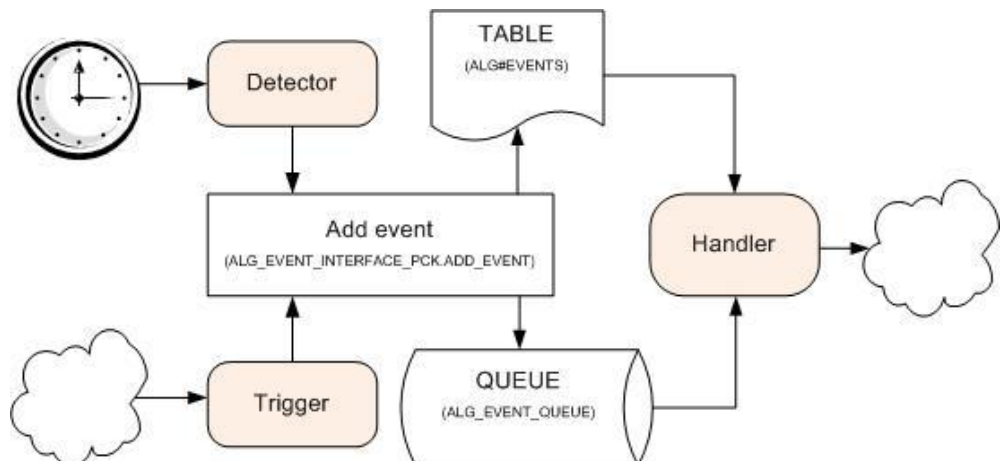


Figure 21: Flows of an event process

6.4.1. Detected Events, Storing to a Table

This event definition is suitable when there is no urgency to act on specific events and individual data changes are not important. For example, there is an event that produces an overview of all policies modified in the previous week. A record of all the individual modifications does not have to be kept. A check on the last date the record was updated is sufficient in this example.

Detected events are also the only events able to act on situations not triggered by data manipulation but the passing of time. For example, a member reaches 18 years of age or a record having a specific status for a number of days. Triggered events are not suitable for this since no data is changed and therefore no database trigger will signal the event.

6.4.2. Triggered Events, Storing to a Table

This event definition is suitable when the action of the event has no urgency but the individual data modifications are important. For example, a triggered event can be used when an event should be registered when a policy reaches the final status. A detected event is less suitable for this because at the time the detection batch is running the policy could have been updated to another status. This results in the policy being skipped by the detection run and no event is registered.

6.4.3. Detected Events, Storing to the Queue

Although technically possible, this type of event is not practical. Detected events are processed in the same batch run. There is not much difference between the moment an event is registered and the moment it is processed. Therefore processing these events using the queue will not provide much of an advantage. The queue will have a large load to process when lots of events are detected.

When multiple occurrences of the same event are required storing to the queue should also be used.

6.4.4. Triggered Events, Storing to the Queue

This event type is best suited when individual updates are important and immediate action is required. For example, the member should receive a welcome email when their policy reaches the final status.

6.5. Framework Components

This chapter describes all the components within the OHI Back Office application for setup, registering and responding to business events.

6.5.1. Event Definition

The Event Definition (SYS1149F) window is used for defining an event in OHI Back Office.

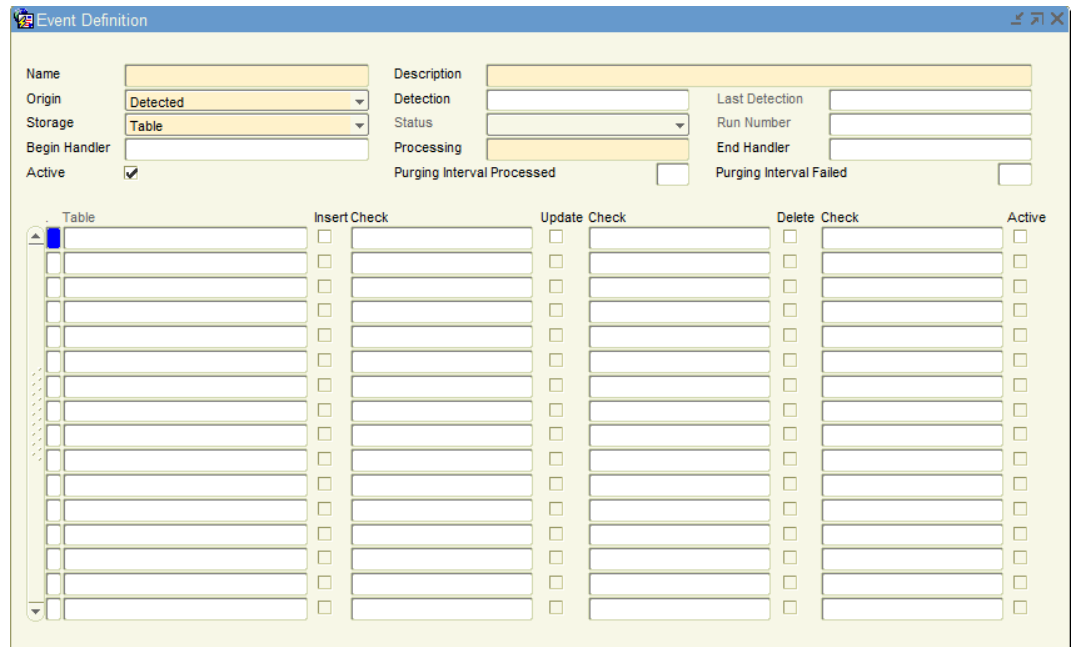


Figure 22: Maintain Event Definitions

Data in Event definitions

Field	Description
Name	The name of the event, maximum length 30 characters.
Description	The description of the event, maximum length 100 characters.
Type	How the event is signaled, allowable values Detected and Triggered.
Detector	The (package) procedure for registering this event. Only applicable for detected events.
Last Detection	The timestamp of the last processing run. Only applicable for detected events.
Storage	Where are signaled events stored, allowed values Table and Queue.
Status	The status of a processing run. Only applicable for detected events.
Run Number	The last number of the processing run. Only applicable for detected events.
Begin Handler	The (package) procedure for the begin handler. Only applicable for events with storage set to Table.
Handler	The (package) procedure for handling the event. Applicable for all events.
End Handler	The (package) procedure for the end handler. Only applicable for events with storage set to Table.
Active	Indicates whether the event is active or not.
Purge Interval Success	The purge interval for events that have been successfully processed. Only applicable for events with storage set to Table.
Purge Interval Failure	The purge interval for events that have failed. Only applicable for events with storage set to Table.

Data in Tables

Field	Description
Table	Holds the name of the table the event is designed for.
Insert	Indicates whether events should be signaled when a new record in this table is created.
Evaluation function	The name of the dynamic PL/SQL definition used to evaluate the event. Only allowed in case the Insert indication is checked.
Update	Indicates whether events should be signaled when a record in this table is updated.
Evaluation function	The name of the dynamic PL/SQL definition used to evaluate the event. Only allowed in case the Update indication is checked.
Delete	Indication whether events should be signaled when a record in this table is deleted.

Field	Description
Evaluation function	The name of the dynamic PL/SQL definition used to evaluate the event. Only allowed in case the Delete indication is checked.

The event tables block is only applicable for triggered events.

The event can be fine-tuned with the evaluation functions to only signal the desired situation. See the next section for a more detailed description of these functions.

6.5.2. Dynamic PLSQL Definition

For triggered events it makes sense to evaluate the contents of the record triggering the event, before deciding whether the event should be registered.

You can create your own PLSQL definition to evaluate the old and new values of the record which triggered the event. You may use the Dynamic PLSQL Definition window (SYS1139F) to register your definition.

The screenshot shows the 'Dynamic PL/SQL Definition' window. The 'Name' field contains 'EVT_CK_ZVP'. The 'Description' field contains 'Decide if an event must be created.'. The 'Scope' is set to 'Event' and 'Check'. The 'Body' field contains the following PL/SQL code:

```

L_rv boolean := false;
begin
    return L_rv;
end;

```

At the bottom, there is a table titled 'Dynamic PL/SQL Tables' with two tabs: 'Single Column' and 'Multi Column'. The table has columns: Table, Column, System Message, Insert, Update, and Delete. The first row has a blue selection bar, and the 'Insert' and 'Update' checkboxes are checked.

Figure 23: Add a trigger condition for an event

The Scope must be set to Event for PL/SQL definitions used within the Business Event Framework. This window is also used to maintain PL/SQL definitions used elsewhere within OHI Back Office. Only the fields applicable for the Business Event Framework are described.

Field	Description
Name	The name of the PL/SQL definition, maximum length 20 characters.
Description	The description of the PL/SQL definition, maximum length 50 characters.
Scope	Should be set to Event to be able to select the definition in the Event Definition (SYS1149F) window.
Body	The actual code of the PL/SQL definition. The event will be registered when the function returns a true value.

The PLSQL Body contains the actual code used to evaluate whether an event should be registered. The code must return a Boolean value to indicate this. In case true is returned the event will be signaled. In case the function returns false, it will not. The old and new values are available as `old_rec` and `new_rec`.

In the following dynamic PLSQL code, an event is only registered if the new status = 'D':

```

/* new_rec and old_rec are set by the calling function */
l_rv boolean := false;
begin
  l_rv :=
  (
    nvl(old_rec.status,'XYZ') != nvl(new_rec.status,'XYZ')
    and
    new_rec.status = 'D'
  );
  return l_rv;
end;

```

Note that the two tabs at the bottom of the SYS1139F window (Dynamic PL/SQL tables - single column) and (Dynamic PL/SQL tables - multi columns) are not used within the context of the Business Event Framework.



Note: 'Active' indication

When committing a dynamic PL/SQL definition with the indication 'Active' checked, OHI Back Office will try to validate the code of the Body section. When creating a new PL/SQL definition the table that will be used is unknown to OHI Back Office. Therefore the 'Active' indication should not be checked when first creating the PL/SQL definition. After linking it to a table in the Event Definition (SYS1149F) window it can be turned on.

6.5.2.1. Passing old_rec and new_rec to your custom developed packages

If you want to pass `old_rec` and `new_rec` to a custom packaged function you should pass them as

`API_<APPLICATION_SYSTEM>_<TABLE_ALIAS>_PCK.API$ROW_TYPE`

For example:

```

/* Compare two GEB_ZORG_VOORNEMEN_PERIODES */
function test_event
(
  p_zvp_old in api_geb_zvp_pck.api$row_type
  , p_zvp_new in api_geb_zvp_pck.api$row_type
)
Return boolean;

```

As said before, the dynamic code is executed under the OHI_DPS_USER account. This means that any custom developed packages called by your code should be granted to OHI_DPS_USER and must have a public synonym (or be prefixed with the schema name).

6.5.3. Event Definition Package

The ALG_EVENT_INTERFACE_PCK can be used to define event definitions. This offers the same functionality as the OHI Back Office Event Definition window with the exception of defining the tables for a Triggered event. The functionality for installing and de-installing an event is available for backward compatibility.

The package also holds procedures that are used for event handling and several utilities.

See Appendix B for a full description of the parameters for each procedure and function in the package.

Event Definition

- **Install**
Available as a procedure and a function returning the ID of the event. This can be used for the event definition. When the given event already exists (based on the name of the event) it will update the event definition, otherwise a new event definition will be registered with the values supplied.
- **De-install**
This procedure is available twice. Once to remove an event with a given name and once to remove it based on the ID of the event definition.

Event Handling

- **Add_event**
Three procedures with this name are available to store an event to the table. One receives the name of the event as a parameter, the second the ID of the event definition. The parameter code holds the identification of the record in OHI Back Office that caused the event. The third procedure stores an event to the Business Event Framework queue. It receives one parameter of type:

`ALG_EDE_PAYLOAD_TP`

To be able to change the storage clause of an event from table to queue the code should be a string with the following format:

`table_id##record_id##dml_type`, where `dml_type` can be 'I' (Insert), 'U' (Update) or 'D' (Delete)

- **purge_all_events**
This procedure is available twice, based on the name of the event definition and based on the ID of the event definition. It will remove all events and event errors for the given event.
- **reapply_failed_event**
This procedure is also available twice, based on the name of the event definition and based on the ID of the event definition. It will change the status of a event stored in the table from 'Failed' to 'New'. This procedure should be called from within the detector plugin. Providing a specific event will reset only the provided event for the given event definition. When no event is provided all failed events for the given definition will be reset.

Utility

- **type_payload_to_code** Can be used to transform object type `alg_edc_payload_tp` to the code parameter of the `add_event` procedure.
- **code_payload_to_type** Available twice, used to convert the code parameter of the `add_event` procedure to object type `alg_edc_payload_tp`. Available with the name and the ID of the event definition.

6.5.4. Event Handling Package

Events are handled by the framework package ALG_EVENT_PCK. This is an internal OHI Back Office package and is therefore not available for custom development. It contains the same functions and procedures as the ALG_EVENT_INTERFACE_PCK.

6.5.5. Process Business Event Batch

The Process Business Events (SYS5001S) batch has been developed to support starting the Business Event Framework by the OHI Back Office batch scheduler. The batch is needed to signal Detected events and to process events which are stored in the ALG#EVENTS table. The batch can be scheduled using OHI Back Office Submit Batch Request (SYSS003F) window. It has the name of the event as a parameter allowing for different run intervals per defined event.

6.5.6. Background Process

Background process OHI_EVENT_JOB_x is used to handle events with storage set to queue. The process is started and stopped simultaneously with the OHI Back Office batch process.

The process monitors the Business Event Framework queue. Events are taken from the queue and processed using the ALG_EVENT_PCK package.

With the Back Office parameter 'No. of processes for event framework' the number of processes listening to the event queue can be set.

6.6. Developing Your Own Business Events

First the business event should be analyzed to determine the best suited registering and handling types. Triggered events are best suited when the event signals data manipulation and it is important to signal each individual action. Detected events can be used for end-of-day status reports or for events not caused by data changes but by the passing of time.

The storage of the event should be set to Queue when, as soon as the event is signaled, immediate action is required. It can be set to Table when the action to the event is less urgent and can occur at a scheduled times.

6.6.1. Detected events

In the Event Definition window set the Type to Detected. The Detector field is mandatory for this type of event.

6.6.1.1. Detector

The field holds the (package) procedure, which is used to register the business event. The procedure receives the timestamp of the last time it was started and the name of the business event. The Business Event Framework will commit after executing the detector.

For example where an event should count the number of policies, the detector in the event definition could be:

```
my_event_pck.detect_nr_policies
```

The procedure definition could look like:

```
procedure detect_nr_policies
( pi_event_name in alg_event_definitities.naam%type
, pi_start_date in date
);
```


Each event occurrence can be stored using the `add_event` procedure in the `ALG_EVENT_INTERFACE_PCK` package.

6.6.1.2. Adding events

Detected events should either be saved to the `ALG#EVENTS` table or to the Business Event Framework queue. This can be done by calling the `add_event` procedure in the `ALG_EVENT_INTERFACE_PCK` package.

Dependent on the storage clause for the event the appropriate `add_event` can be called. For events stored in the table this would be:

```
alg_event_interface_pck.add_event
( pi_name in alg_event_definities.naam%type
, pi_code in alg#events.code
, pi_date in alg#events.master_date%type
);
```

Or:

```
alg_event_interface_pck.add_event
( pi_ede_id in alg_event_definities.id%type
, pi_code in alg#events.code
, pi_date in alg#events.master_date%type
);
```

For events stored in the queue this is:

```
alg_event_interface_pck.add_event
( pi_ede_payload in alg_ede_payload_tp
);
```

If the storage type of an event is modified in the Event Definition (SYS1149F) window the `add_event` will continue to work and the received parameters will be converted to match the storage type. Although it can have a (minor) impact on performance it is not necessary to change the detector-program code.

6.6.1.3. Example

The following code shows an example of an event to signal all new relations created since the last time this event was processed.

```
procedure my_detector
( pi_event_name in alg_event_definities.naam%type
, pi_start_date in date
) is
  cursor c_events
  ( vi_date_from date
  ) is
    select rel.id
    from rbh_relaties rel
    where rel.creatie_moment >= c_events.vi_date_from
    ;
  l_tab_id alg_tabellen.id%type;
  l_dml_type varchar2(1) := 'I';
begin
  -- Determine table id
  l_tab_id := rbh_rel_capi.g_tab_id;
  for r_rec in c_events ( pi_start_date )
  loop
    -- Store to a table
    alg_event_interface_pck.add_event
    ( pi_name => pi_event_name
    , pi_code => r_rec.id
    );
```

```
    end loop;  
end my_detector;
```

6.6.2. Triggered events

In the Event Definition window set the Type to Triggered. For events of this type the Detector field is not available since the event is signaled using OHI Back Office database triggers.

6.6.2.1. Tables

The second block is only available for triggered events. The table of the event can be defined and the action on the table can be set using the Insert, Update or Delete indications.

6.6.2.2. Evaluation

Evaluation functions are available for defining additional criteria for registering an event. These functions can be set up in the Dynamic PL/SQL Definition window.

6.6.2.3. Example

A triggered event can be set up to signal all policies that reach a final status. Since policies cannot be created with the final status, the only action to monitor is update. To prevent registering other updates to the policy the following dynamic PL/SQL can be created, the scope of the dynamic PL/SQL should be set to Event. The body can hold:

```
-- declaration section: define cursors and variables here  
  l_retval boolean := true;  
begin  
-- body section:  
-- return boolean value --  
return new_rec.status = 'D'  
  and new_rec.status <> old_rec.status;  
end;
```

Since this PL/SQL definition will be linked to the VER_POLISSEN table in the event definition window the new_rec and old_rec variables will hold all fields available in that table.

6.6.3. Batch Handled Events

Events with storage clause set to table will be handled by the Process Business Events batch.

6.6.3.1. Begin Handler

The (package) procedure defined for the begin handler in the event definition is called once. This can be used for example to open a file for writing log messages. The framework will commit after executing the begin handler.

The (package) procedure receives the following parameters:

- The name of the event
- The run number of the process
- The date of the last processed run

```
my_event_pck.begin_handler  
( pi_name in varchar2  
, pi_run_nr in number  
, pi_date_detection in date  
);
```

6.6.3.2. *Handler*

The handler is called for each instance of the event. The framework will commit after executing the handler. The handler (package) procedure receives the following parameters:

- The code of the event
- The date that was passed when registering the event
- The date the event was registered

```
my_event_pck.handler  
( pi_name in varchar2  
, pi_date_source in date  
, pi_date_detection in date  
);
```

6.6.3.3. *End handler*

The end handler is called once after processing all events. For example this can be used to save information about the process run such as the total number of events processed, the number of failed events or close the file opened in the begin handler. The framework will commit after executing the end handler.

```
my_event_pck.end_handler  
( pi_name in varchar2  
, pi_run_nr in number  
, pi_date_detection in date  
);
```

6.6.3.4. *Purge intervals*

Purging old event records is a batch function. It is possible to set up the intervals in the Event Definition window. It is possible to have different values for failed events since investigation may take longer than successful events.

The batch will remove records from the ALG#EVENTS tables at the end of the run.

6.6.4. **Near Real Time Events**

Events stored to a queue are processed by a continuous Background Process. Since each event is processed individually no begin handler or end handler is available for these events. Only the handler is applicable.

6.6.4.1. *Handler*

The OHI Back Office event package will take an event from the queue and call the handler defined in Event Definition window. The (package) procedure for this handler receives an object as parameter. This object contains the following information.

- The ID of the event definition
- The ID of the table the signaled record is stored in
- The record ID
- The DML type that caused the event

The handler can be defined as:

```
my_event_pck.queued_event_handler  
( pi_load in alg_edc_payload_tp  
);
```

This handler determines of course what is done with the event. In the situation that publishing the even through a JMS queue is a requirement this can be done as described in the Appendix regarding the [use of a JMS messaging queue](#).

6.6.5. Custom Plug-ins

The event tables and event framework are pre-installed in the OHI Back Office database. The custom plug-ins for the detector and handlers of the events must be implemented in a separate database schema.

The following is assumed for the purpose of this installation procedure:

- The event definition is called my_event
- The custom components are combined in a single package called my_event_pck
- The my_event_pck.install procedure creates and configures an event definition for my_event
- The OHI components are owned by database schema ozg_owner
- The database schema for bespoke software is called my_schema
- The business event framework is started by the ozg_batch schema. However all dynamic PLSQL code is executed under OHI_DPS_USER.

The installation consists of the following steps:

- Ensure that public synonyms and access privileges are created for the ozg_owner components that are accessed by the my_event_pck package (you may only use the objects granted by the \$OZG_BASE/OZG_DIRECT.grt sqlplus script for this; consult the Object Authorization manual for how to use this script)
- Ensure that my_schema has execute privileges for ozg_owner.alg_event_interface_pck (should be taken care of in the previous step but in previous releases the grant was missing)
- Compile the package specification and package body for my_event_pck under the database schema my_schema
- Create a public synonym my_event_pck for my_schema.my_event_pck
- Grant execute privileges for my_schema.my_event_pck to the OHI_DPS_USER schema.
- Run my_event_pck.install under my_schema to install the definition for my_event or set up the event definition using the Event Definition window.

6.7. Processing Business Events

Processing business events is dependent of the business event definition. Detected events are registered by the Process Business Events batch. Triggered events are started by database triggers.

6.7.1. Process Business Events Batch

The Process Business Events (SYS5001S) batch has been developed to start up a business event processing run. The batch can be scheduled using the Submit Batch Request (SYSS003F) window.

Figure 24: Sample of scheduling Process Business Events (SYS5001S) batch

In the screenshot business event 'AZR_MOD_PRT' will start every hour.

The batch serves two purposes and is only needed for these types of events:

1. Signal Detected events. For detected events the program code defined by the Detector is executed once. See Detector for a more detailed description of the detector. This step is skipped for triggered events.
2. Process events stored in a table. Events stored in the table are processed. First the specified Begin Handler is called once. Per event the Handler is called to process the event. After processing all events the End Handler is called once. After the end handler the batch purges old events. This step is skipped for events stored in the queue.



Note: Detection and Processing in one run

For Detected events storing the events to a table and registering and processing the events happen in the same processing run.

6.7.2. Queued Events

Events stored to the queue are processed by a dedicated process monitoring the business event queue. Events are taken from the queue and the handler is called to process the event.

The dedicated process is started and stopped together with the OHI Back Office batch scheduler.

6.8. Examples

This chapter contains examples of how to set up events to be handled by the Business Event Framework.

6.8.1. Detected Event, Store to a Table

This example shows an event to signal all relations that have been updated since the last time the event was run. It writes the identification of the relations to a file. Since there is no need to act on individual updates and no immediate action is required upon the change, a detected event storing to a table will suffice.

6.8.1.1. Event definition

The screenshot shows the 'Event Definition' window with the following configuration:

- Name: OHI_DEMO_D_T
- Description: Example detected to table
- Warning: Detected
- Surcharge: Table
- Begin Handler: OHI_EVENT_DEMO_PCK.START_HANDL
- Active:
- Detection: OHI_EVENT_DEMO_PCK.DETECTI
- Status:
- Processing: OHI_EVENT_DEMO_PCK.HANDLE
- Purging Interval Processed: 10
- Purging Interval Failed: 32
- Last Detection:
- Run Number:
- End Handler: OHI_EVENT_DEMO_PCK.END_HA

The table below defines the event triggers:

Table	Insert Check	Update Check	Delete Check	Active
	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Figure 25: Example event definition named OHI_DEMO_D_T

Example OHI_DEMO_D_T is a detected event (Type is set to Detected) and it will store events to the ALG#EVENTS table (Storage set to Table).

6.8.1.2. Detector

The detector of the event in this example is OHI_EVENT_DEMO_PCK.DETECTOR_T. Typically the code of a detector consists of a query to select the records to be registered and a call the ALG_EVENT_INTERFACE_PCK package to save the identification of the selected records:

```

procedure detector_t
( pi_name      in varchar2
, pi_date_from in date
) is
begin
    for l_rec in ( select rel.id
                  ,      rel.laatste_mutatie_moment
                  from  rbh_relaties rel
                  where rel.laatste_mutatie_moment >=
pi_date_from
                )
    loop
        alg_event_interface_pck.add_event
        (pi_name => pi_name
        ,pi_code => to_char(l_rec.id)
        ,pi_date => l_rec.laatste_mutatie_moment
        );
    end loop;
end detector_t;

```

6.8.1.3. Begin handler

The begin handler OHI_EVENT_DEMO_PCK.START_HANDLER_T in this example is used to open the file for writing the identifications of the relations.

```

procedure start_handler_t
( pi_name          in varchar2
, pi_run_nr       in number
, pi_date_detection in date
) is
  l_filename      varchar2(100);
  l_location      varchar2(100) := 'OZG_TMP';
  l_max_linesize  constant binary_integer := 32767;
begin
  l_filename := i_name||
               '_'||
               to_char(pi_run_nr)||
               '_'||
               to_char(pi_date_detection, 'YYYYMMDDHH24MISS')||
               '.txt';
  a_file_handle := utl_file.fopen
                  ( l_location
                  , l_filename
                  , 'w'
                  , l_max_linesize
                  );
end start_handler_t;

```

6.8.1.4. *Handler*

The handler OHI_EVENT_DEMO_PCK.HANDLER_T of the example writes the data to the opened file. The code parameter contains the identification of the relation record. It could be used to select more detailed information from the relation record. For instance who and when the last modification was made. Since this is a detected event the data would however only reflect the last modification.

```

procedure handler_t
( pi_code          in varchar2
, pi_date_source   in date
, pi_date_detection in date
) is
begin
  utl_file.put_line
  ( a_file_handle
  , pi_code||
    ' -> '||
    to_char(pi_date_detection, 'YYYY-MM-DD HH24:MI:SS')||
    ' -> '||
    to_char(pi_date_source, 'YYYY-MM-DD HH24:MI:SS')
  );
end handler_t;

```

6.8.1.5. *End handler*

The end handler OHI_EVENT_DEMO_PCK.END_HANDLER_T in this example is used to close the file.

```

procedure end_handler_t
( pi_name          in varchar2
, pi_run_nr       in number
, pi_date_detection in date
) is
begin
  if utl_file.is_open(a_file_handle)
  then

```

```

        utl_file.fclose(a_file_handle);
    end if;
end end_handler_t;

```

6.8.1.6. Scheduling the event

All steps for the detected event have finished. The event can be scheduled to run using Submit Batch Request (SYSS003F) window.

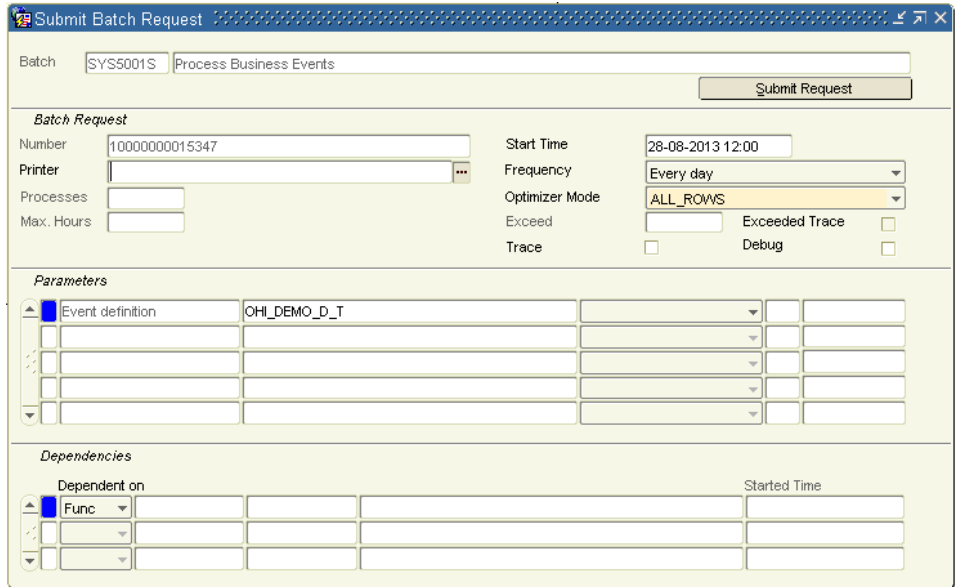


Figure 26: Submit a batch request to run event OHI_DEMO_D_T

6.8.2. Triggered Event, Store to the Queue

This example shows how to define an event that will be registered when a new record is created.

6.8.2.1. Evaluation Function

First step is creating the evaluation function which will be used in the event definition. OHI Back Office will validate the entered PL/SQL code when it is saved. Since the PL/SQL Definition has not yet been linked to a table this validation will fail. Therefore the indicator 'Active' should not be checked. In that case the PL/SQL code is disabled and will not be validated.

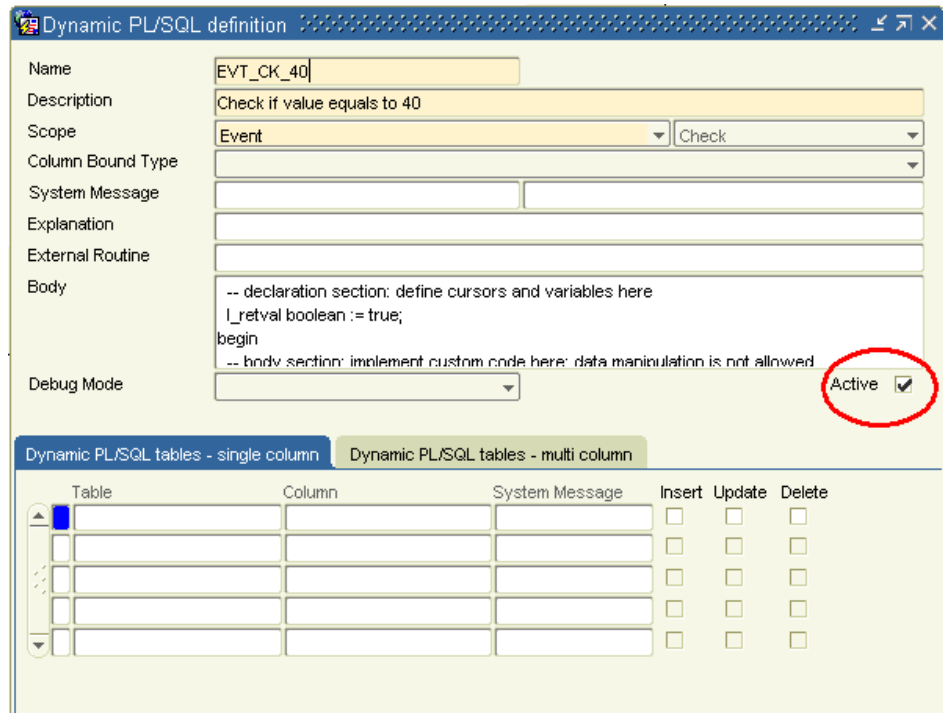


Figure 27: Event evaluation function activation

The complete Body of the PL/SQL definition is not visible, it contains:

```

cursor c_rel
( v_rel_nr in rbh_relaties.nr%type
) is
select rel.n_geslacht gender
from   rbh_relaties rel
where  rel.nr = c_rel.v_rel_nr
;
l_rel_gender rbh_relaties.n_geslacht%type;
l_retval    boolean := true;
begin
open  c_rel( v_rel_nr => new_rec.rel_nr );
fetch c_rel
into  l_rel_gender;
close c_rel;
l_retval :=      l_rel_gender = 1
               and new_rec.code = '40'
return l_retval;
end;

```

The example shows that when a record is created for a male relation and the code is equal to '40' the function will return true and an event will be registered.

Since it is linked to the RBH_DERDEN_CODERINGEN table in the event definition, the new_rec will hold all the new values of the record. It can be used for more sophisticated evaluation than this example.

6.8.2.2. Event definition

The screenshot shows example event definition named OHI_DEMO_T_Q. It is a triggered event (Type is set to Triggered) and it will store events to Business Event Framework queue (Storage set to Queue). This type of event will be signaled by database triggers so a separate Detector is not needed. A begin handler and end handler are not required when the storage clause is set to Queue as the events will be handled.

Figure 28: Add a table and evaluation function to a triggered event

The event will be signaled by the creation of a record in the RBH_DERDEN_CODERINGEN table when the dynamic PL/SQL definition EVT_CK_40 returns true.



Note: 'Active' indication

After the PL/SQL Definition has been linked to a table the 'Active' indication in the Dynamic PL/SQL Definition window must be checked to validate and enable the code.

6.8.2.3. Handler

The events are handled by OHI_EVENT_DEMO_PCK.HANDLER_Q. It takes the identification of the record from the object type it receives as parameter and sends an email to notify another department for instance.

```

procedure handler_q
( pi_load in alg_edc_payload_tp
) is
  l_conn utl_smtp.connection;
  l_code varchar2(100);

  procedure send_header
  ( pi_name in varchar2
  , pi_header in varchar2
  ) is
  begin
    ult_smtp.write_data
    ( l_conn
    , pi_name || ': ' ||
    pi_header || UTL_TCP.CRLF
    );
  end send_header;
begin
  l_code := alg_event_interface_pck.type_payload_to_code
    ( pi_edc_payload => pi_load
    );

  l_conn := utl_smtp.open_connection('smtpsrv.mycompany.com');
  utl_smtp.helo( l_conn, 'mycompany.com');

```

```

utl_smtp.mail( l_conn, 'sender@mycompany.com');
utl_smtp.rcpt( l_conn, 'recipient@mycompany.com');
utl_smtp.open_data(l_conn);

send_header('From', '"Sender" <sender@mycompany.com>');
send_header('To', '"Recipient" <recipient@mycompany.com>');
send_header('Subject', 'Relation created');

utl_smtp.write_date( l_conn
                    , UTL_TCP.CRLF ||
                    'Relation with code ' || l_code ||
                    ' has been created. '
                    );
utl_smtp.close_data( l_conn );
utl_smtp.quit( l_conn );
end;

```

6.9. Trouble shooting the event framework

Sometimes a triggered and or queued event seems not to do anything. The following list gives some handhels where to look for.

- Is the batch scheduler running? The event processes are monitored by the batch scheduler. If the batch scheduler is not running the event processing jobs are probably neither.
- Is the value for the Back Office Parameter 'No. of processes for event framework' not set to a value less or equal to 0? This parameter set the amount of active workers to process the queue.
- Are the worker jobs running? Check as owner in user_scheduler_jobs for active/running jobs with a name like 'OHI_EVENT_JOB_n_P', where n is a numeric value.
- Is the event active? Make sure the active indicator both at event level as well as table level are activated. Also a check condition when being defined should be active.
- Is the handler package granted to the OHI_DPS_USER scheme of OHI Back Office? This will be the user executing the call defined in the handler.
- Look into the table ALG#EVENT_ERRORS to see if there are issues when the process tries to execute the handler function.

7. Custom Batch Scripts

OHI Back Office allows you to create your own batch processing scripts and run them with the standard batch scheduler. Batches can be purely PL/SQL or they can be Perl or OS Shell scripts. They can be scheduled to run immediately or at a predetermined time.

For PL/SQL batches you are advised to use the built-in OHI Back Office script generator to generate a framework to control the execution of your custom code, especially if you expect to be processing large data volumes. The generated framework helps to process large volumes in manageable chunks and provide the same kind of feedback as the standard scripts.

7.1. Approach

A custom batch script is handled in the same way as a standard script.

This means that:

- Most batch scripts process a large amount of data, and report results, often after performing DML operations on the selected data.
- A custom batch script must be registered as a script
- A custom batch script is run under the BATCH account
- While creating a request for a custom batch script, the end user may enter script parameters.
- At runtime the custom batch script may evaluate the script parameters.
- When a custom batch script completes or aborts it must register its end state in the batch requests table.
- A script must be created to export the batch definition from the test database to the production database.

The typical flow for a batch script looks like this:

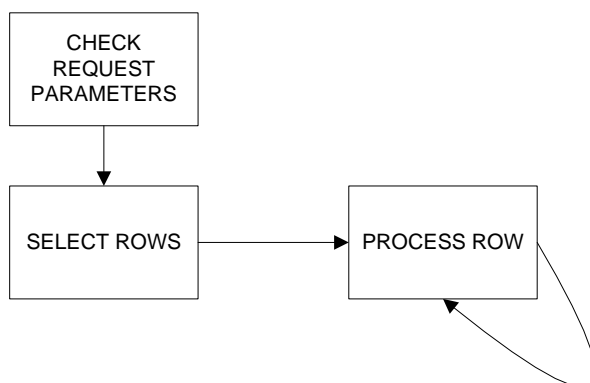


Figure 29: Typical process for a batch script

7.1.1. Create or generate?

Although you may want to manually create all code for your batch script, you are strongly advised to start with a template created by the generator built into OHI Back Office and modify this to your needs.

The generator (SYS_GEN_PCK) creates a framework to process the data in small chunks with multiple parallel processes.

The framework helps you to define the data to be processed by defining the selection of the data, the so called process units, and process the data chunk by chunk which usually is record by record over multiple threads.

Note that the custom batch scripts based on these templates provide the same throughput statistics as standard batch scripts created by OHI Back Office.

The generator supports the following batch types:

- Bulk DML processing batches with support for parallel execution
- CSV file export
- CSV file import
- XML file import

7.2. Batch user

Note that the batch scheduler connects to the OHI Back Office database with the database user, usually named BATCH. This is not the user who created the batch request! In the rest of this document we assume the batch account is named BATCH but as the name may be chosen when implementing OHI Back Office it may be different.

The user BATCH must be able to access the database objects needed to process the batch request. For OHI object access this is arranged through role OHI_ROLE_BATCH which is granted the minimum set of privileges on OHI objects to prevent potential misuse of the privileges of the batch account.

The database objects for custom batches reside in a custom schema, for example SVS_OWNER. The BATCH user must be granted access to these database objects, by means of grants to the user itself or to the role OHI_ROLE_BATCH.

7.3. Registration

7.3.1. Batch

The definition of a batch is maintained in the screen "Batch" (SYS1008F). In this screen you can define or check the main batch characteristics, like the batch code, whether it can spawn parallel processes, whether it is a PL/SQL, Perl or OS shell script etc.

Figure 30: Batch definition

Also the batch parameters can be defined or checked here. Parameter definitions can be shared between different batches, which means that if one is to use an already defined parameter, care has to be taken that the definition of the parameter is not inadvertently changed.

Parameters are always grouped in parameter sets. E.g. the location of a file might be specified by two grouped parameters: the directory and the file name. Most of the time, however, a parameter set will contain just one parameter.

Parameter sets have a few checks. If provided the values may only be chosen from a list of values, maybe with additional restrictions. They can also be checked using a parameter set validation.

Also a list of values can be set to be used. The restriction on the list of values depends on the particular list of values. It is best to look for the use of an actual list of values to see how it is used and how it can be restricted. Besides the predefined lists of values also a dynamic list of values can be used.



Note: The LOV for the dynamic value list is SYS2019L and the LOV restriction should contain the name of the dynamic value list. The column field at last contains the sequence number of the column in the dynamic value list..

Figure 31 shows the screen for defining parameter set validations. It defines a validation as it might be used for selecting relations that are physical persons.

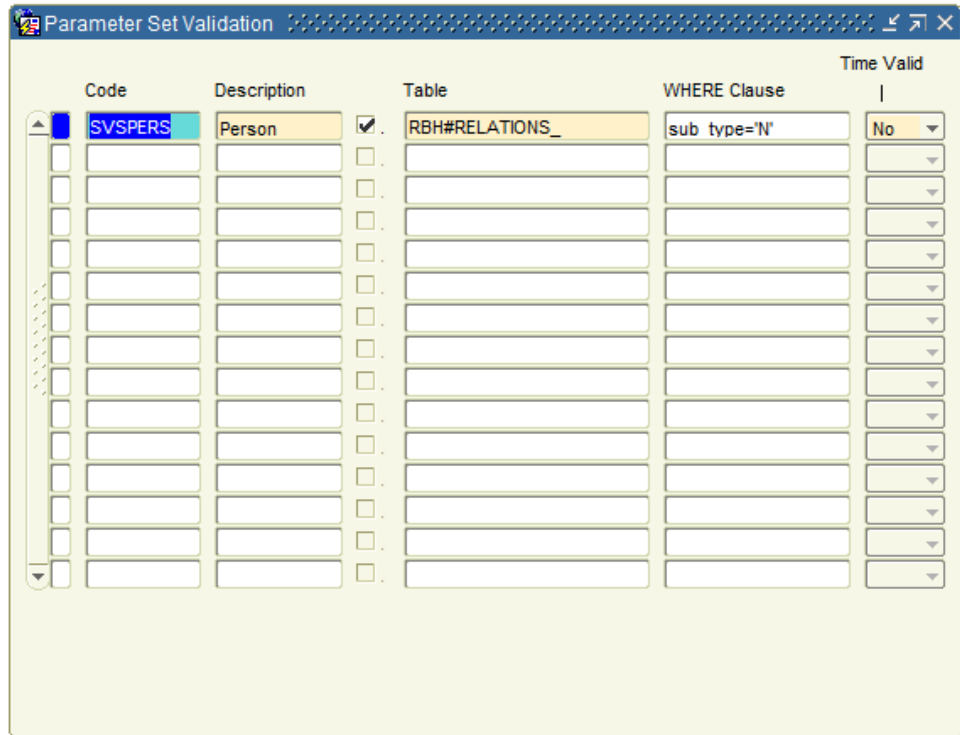



Figure 31: Defining a validation on the type of relation.

Relations in OHI can be either persons or organizations. Both are stored in the same table and are distinguished based on the sub type. The parameter set validation here is called SVSPERS. The checkmark tells OHI that it is based on a table, which is given in the field Table. To make sure that the relation is a person (the actual validation) a where clause is specified, that checks the sub_type which must be 'N' for (natural) persons.

7.3.2. Dynamic List of values

In the screen "List of Values" (SYS1152F) a custom list of values can be defined. This list of values uses tables of OHI Back Office. If the use of a custom table is required than this table should be used in a "System view" (SYS4001F). This system view can be used as the "table" of the list of values.

 **Note:** The table or (system) view used as table source in the list of values should also have one numeric column named ID. It is not required to use the ID column as a column in the list of values.

7.3.3. Batch Settings

For batches created with SYS_GEN_PCK a couple of settings are important.

- Code and description: The code is the identifier of the batch and should start with SVS
- Batch type: defines the type of batch and should be
 - SQL*Plus for standard mutating batches and CSV report files
 - CSV for incoming CSV files
 - XML for incoming XML files
- New Standard? Indicator must be activated

- Required parameters: Some batch scripts do have a set of required parameters. See SYS_GEN_PCK for the required set of parameters per type of script.
- The value of the field “Name in script” at the parameter set component should be the same as used in the definition of the parameter in the call to SYS_GEN_PCK

7.4. Export script definition

The definition of a batch as defined in the screen “Batch” (SYS1008F) can be extracted to an installation script that can be used to install this definition in other OHI environments, e.g. test or production.

The procedure write_module_ins_file in SYS_GEN_PCK takes care of this.

```
Sys_gen_pck.write_module_ins_file
( pi_module      => 'SVS0001S'
, pi_file_location => 'OZG_TMP'
);
```

The file will be placed in the database directory provided at the parameter pi_file_location and is called the provided module code (pi_module) with the extension .ins.

Besides the batch module also the system message will be present in the file as long as the message(s) starts with the module code.

7.5. Generator: Bulk Processing Batch - Overview

The procedure SYS_GEN_PCK.BATCH_SOURCES can be used to generate pre-defined templates for batch processing.

The generation results in a couple of files:

- MODULE.sql
- MODULE.ins
- MODULE_PCK.pck
- MODULE_CUST_PCK.pck

7.5.1. MODULE.sql

The sql file will be called by the batch scheduler when a batch request is created for this module. It will start the batch by executing the run procedure in the MODULE_PCK package. This file should not be changed and be placed in the sql directory of the application server of the environment.

7.5.2. MODULE.ins

The ins file contains data that is used by some system messages and should be executed in a sql session in the database.

NB. This is not the final module installation script, see chapter 7.11 Module installation script for that.

7.5.3. MODULE_PCK.pck

This package contains all the generic code to run a batch. This file should not be modified and should be compiled in the database.

This package takes care of the initialization of the batch, the transaction and exception handling and makes callouts the MODULE_CUST_PCK package.

7.5.4. MODULE_CUST_PCK.pck

This package contains all the custom code of the batch and is the one source that must be provided with the required business logic for this batch.

7.5.5. Basic PL/SQL Batch flow

A PL/SQL batch generated with SYS_GEN_PCK typically follows the flow as shown in figure x

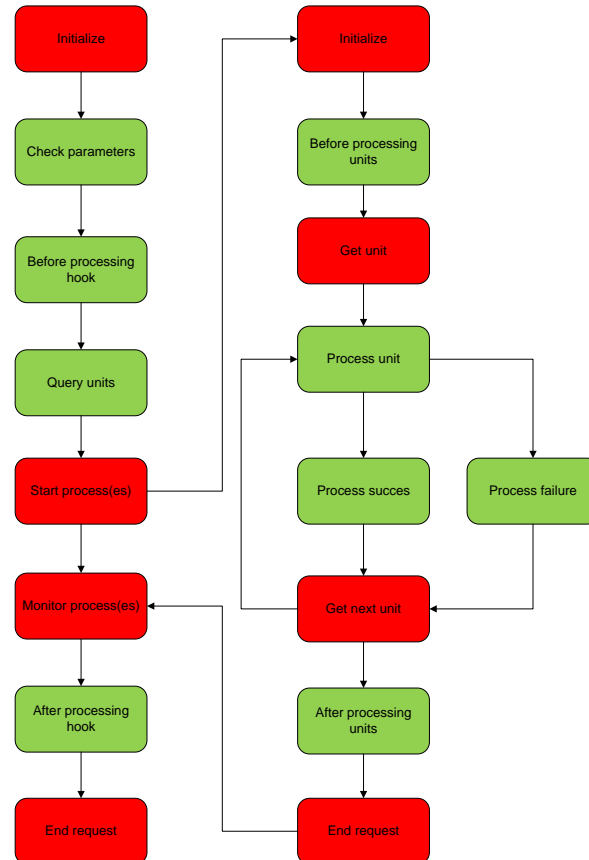


Figure 32: batch process flow

The green blocks contain the batch specific code and have to be implemented. These units are present in the MODULE_CUST_PCK. The red units contain the generic batch code and are present in the MODULE_PCK.

7.6. Generator: Bulk Processing Batch - Details

7.6.1. Define the batch definition

A (bulk) processing batch first selects rows for processing and then processes each row:

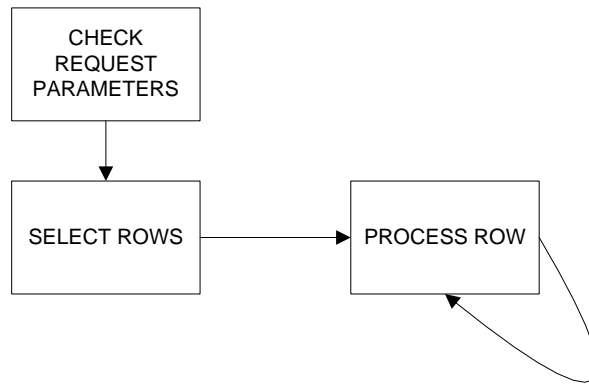


Figure 33: Select and process rows

Before designing the batch you should decide:

What operation should be performed on which data?

The 'which data' determines your selection criteria, the 'what operation' determines the processing.

Note:

- The simplest version is a selection of rows on which you directly perform the DML.
- It may also be that the DML action triggers an update to a master record. In that case you would like to handle all the details for the same master in the same session/sub-process to avoid locking issues. In this second case, you also have to ask whether you want that is committed when all details are processed or that you want to commit per detail itself.
- A variation on this is that you want to perform two different actions with the same master. The second action is determined by the result of the first action.
- A variant of 1 and 2 together is a cursor which is executed per row from the selection.

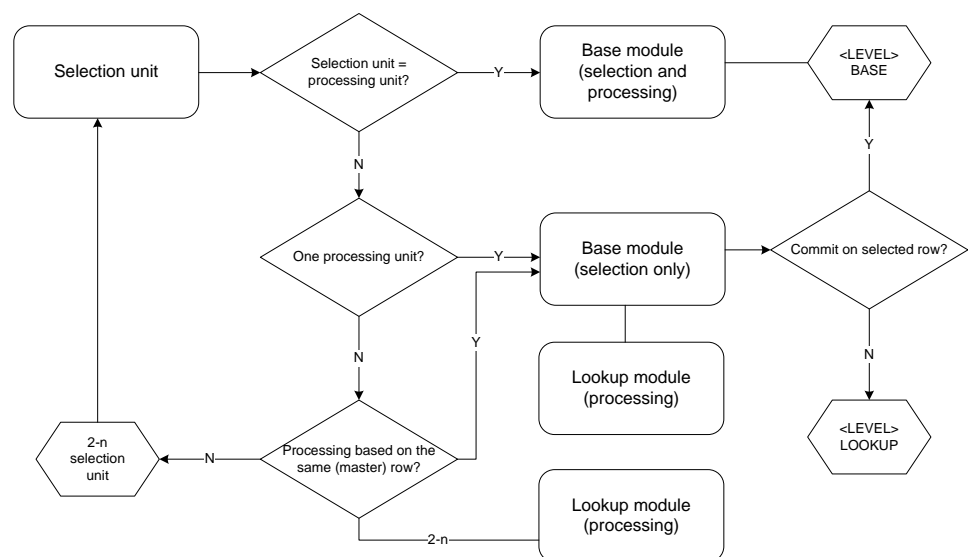


Figure 34: Determine the batch definition setup

The batch template consists of several parts in one call. An example call is available in the package specification of the package SYS_GEN_PCK.

The different parts are:

- The batch module
- 0-n batch parameters
- 1-n batch groups

1-n batch units per group

7.6.1.1. Define the batch module

The batch is the actual module as defined in the screen “Batch” (SYS1008F).

```
sys_gen_pck.batch_sources
( pi_file_location      => 'OZG_TMP'
, pi_module_name       => 'SVS0001S'
, pi_ind_count_processed => true
, pi_parameter_tab     => sys_gen_pck.batch_parameter_tabtype
                        ( sys_gen_pck.batch_parameter_rec
                          ( pi_name => . . .
```

Provide a valid database directory for the parameter `pi_file_location`. This is the directory the files will be created in. The parameter `pi_module_name` contains the code of the batch module as defined in the screen “Batch”. With the parameter `pi_ind_count_processed` set to true a couple of system messages will be created to report when the batch request is completed about:

- the total amount of workunits
- the amount of successful processed units
- the amount of work units with an error

7.6.1.2. Define the batch parameters

The batch parameters are used to influence the behavior of the batch based on the input provided by the user when creating the batch request.

Each batch parameter should be added to the call of `BATCH_SOURCES`

```
sys_gen_pck.batch_parameter_rec
( pi_name      => 'P_MER_CODE'
, pi_table_name => 'VER#POLICIES_'
, pi_column_name => 'MER_CODE'
, pi_data_type  => 'VARCHAR2'
, pi_data_length => 5
, pi_data_precision => 0
, pi_comment   => 'The code of the brand'
)
```

The parameter `pi_name` must have the same name/code as defined in the screen “Batch” in the field “Name in script”.

7.6.1.3. Define the processing unit(s)

The processing unit has two components, the “group” defines the selection and the “unit” defines the processing unit.

```
, pi_group_tab => sys_gen_pck.batch_group_tabtype
                ( sys_gen_pck.batch_group_rec
                  ( pi_table_name => 'VER#POLICIES_'
                    , pi_alias    => 'POL'
                    , pi_label    => 'Policies'
                    , pi_identification => 'NUM'
                    , pi_level    => 'BASE'
                    , pi_alias_fu  => 'POL'
                    , pi_unit_tab  => sys_gen_pck.batch_unit_tabtype
```

The group implements the selection part of the batch. The provided table and column are used to create a basic query which can be adjusted later on. The alias (parameter `pi_alias`) is used to identify the group throughout the batch and should be unique within the batch. The parameter `pi_level` can have one of the two values “BASE” or

“LOOKUP” and is used for the transaction level. The label (pi_label) is used in the messages created by the batch to report on the processed amount of work.

The parameter pi_alias_fu needs some explanation, sometimes the workload defined in either the group or the unit is not the actual workload to be reported, e.g. when the batch uses a grouping of data to be processed. A function will be provided to record the actual amount of data per group of unit when applicable. To use this option the alias of the parameter pi_alias_fu should be a different one than the alias of the group of unit.



Note: The group expects a unique single numeric value per processing unit, e.g. a record id, a policy number or a relation number (parameter pi_identification).

```
, pi_unit_tab => sys_gen_pck.batch_unit_tabtype
  ( sys_gen_pck.batch_unit_rec
    ( pi_table_name => 'VER#POLICIES_'
      , pi_alias      => 'POL'
      , pi_label      => 'Policies'
      , pi_alias_fu   => 'POL'
    )
  )
```

The unit implements the worker process. It can be the same unit as defined at the group or another unit when within the defined group a more detailed processing is needed. If a unit is based on another alias (pi_alias) a second query will be generated to transform the selected record from the group into a more detailed record structure for the unit.



Note: One batch can exist of one or more groups and each group exists of one or more units. Although one group with one unit is the most common type.

7.6.2. Implementation

When the sources are generated the actual implementation can be done. The <MODULE>_CUST_PCK custom package is the only source that needs to be modified for the implementation.

The custom package has several functions and procedures. For "_xyz" the alias you defined for your batch group or unit should be substituted when reading this.



Note: Don't add transaction code like rollback or commit in your custom implementation as well as a “when others” exception handler as this is dealt by the main package.

See Figure 32 for a schematic flow of the different functions and procedures and their place in the process

7.6.2.1. Function get_revision

This function determines the revision of the batch as it is shown in the .out file produced by the batch request.

It is called once in the main batch request

7.6.2.2. Function parameters_ok

In the function parameters_ok the script parameters can be validated.

Using batch request validation is preferred over implementing validation here as it prevents the batch request from being submitted. If any value is not correct a

message can be written and the boolean value "false" is returned. This causes the batch to stop and the status "Error" is recorded.

It is called once in the main batch request

7.6.2.3. *procedure before_processing*

The purpose of this procedure is to execute specific actions once when the batch is started. For example initialization or creating a master record to be referred to by details that are generated during batch processing.

It is called once in the main batch request

7.6.2.4. *function query_xyz*

The query_<alias> function prepares the processing units.

The parameter pi_volgnr in the procedure alg_batch_pck.ins_svh_bulk determines the order in which the units are processed.

If more than one subprocess is started process (1) will handle row 1 and process 2 will handle row 2 etc.



Note: Implementing an 'DUP_VAL_ON_INDEX' exception handler will not work as the inserts in the table will be done in bulk and the last set will be done post processing the function 'query_xyz'. When, in the exceptional case you need to trap the 'DUP_VAL_ON_INDEX' please add a call at the start of the function to 'ALG_BATCH_PCK.SET_IGNORE_DUPVAL'.

It is called once in the main batch request

7.6.2.5. *procedure after_processing*

The purpose of this procedure is to execute specific actions once when the batch is finished. For example cleaning up temporary data, generating a message based on the manual counts, etc.

It is called once in the main batch request

7.6.2.6. *cursor a_c_xyz*

In the package specification a cursor can be defined. This will be the case if a Structure is provided with a different unit compared to the group.

The query_xyz function queries the rows from the base usage and during processing a single row is presented to this cursor (v_record_id).

Using this cursor the selection from the lookup usage can be performed. Each row from the resulting set will be presented to the process_xyz procedure. The order of the query result is the order of the rows processed by the process_xyz procedure.

It is called once per record from the group selection in the subprocess batch request.

7.6.2.7. *procedure update_shg*

The procedure update_shg is meant to customize recording of the number of records that was processed successfully or with errors using manual messages.

Using the procedure alg_batch_pck.toevoegen_shg the message is initialized. Updating the number of processed records must be coded manually.

An update is chosen because otherwise too many records would be created causing trouble when purging batch requests.

Updating is done using an autonomous transaction and does not influence processing.

7.6.2.8. *procedure before_process_xyz*

The purpose of this procedure is to execute specific actions once when the group is started. For example initialization of a package variable.

It is called once per group in the subprocess batch request

7.6.2.9. *procedure after_process_xyz*

The purpose of this procedure is to execute specific actions once when the group is finished. For example cleaning up temporary data, generating a message based on the manual counts, etc.

It is called once per group in the subprocess batch request

7.6.2.10. *procedure process_success_xyz*

This procedure can be used to write a message saying that processing was successful or to maintain a specific counter.

Sometimes an indicator needs to be updated when the transaction was successful. Such an action can be performed here too.

It is called once per record (unit) in the subprocess batch request

7.6.2.11. *procedure process_error_xyz*

This procedure can be used to write a message saying that processing was not successful or to maintain a specific counter.

Transactions are not allowed. Recording or updating manual messages is allowed provided procedure `update_shg` en `alg_batch_pck.toevoegen_shg` are used. These messages are processed using an autonomous transaction.

It is called once per record in the subprocess batch request in case of an error.

7.6.2.12. *procedure process_xyz*

The procedure `process_<alias>` carries out the 'real' processing per processing unit.

In the `process_<alias>` procedure each row can be validated if it is suited for processing. If it is not, an error message can be written. In that case processing the row will be stopped.

It is called once per record (unit) in the subprocess batch request

7.6.3. Messages

Per record a system message can be created for informational purpose or to record an error. In case of an error the current record is skipped from being processed and the next record will be processed.

An example error message will look like

```
sys_message_handling_pck.give_error
( pi_msg_code           => 'SVS0001S004'
, pi_msg_default_text  => 'Record #1# does not comply to the specifications.'
, pi_msg_parmvalue_tab => sys_message_handling_pck.msg_parmvalue_tabtype
                          ( sys_message_handling_pck.parmvalue
```

```

        ( pi_sequence_nr => 1
          , pi_value_number => pi_record_id
        )
      , sys_message_handling_pck.parmvalue
        ( pi_sequence_nr => 2
          , pi_value_char  => 'CHECK001A'
        )
      )
, pi_raise_exception => true
, pi_table_id        => 123
, pi_record_id       => pi_record_id
);

```

Be sure to register the system message in the screen “System messages” (SYS1002F). Also note that each parameter value cannot exceed a length of 120 characters and the total length of a message with the parameters substituted cannot exceed a length of 2000 characters.

7.7. Generator: CSV Export Batch

The procedure `sys_gen_pck.export_csv_sources` creates a template for a script to write the result of a query into a CSV (comma separated value) file. Note that the way the template is created enables translation of the report label and the report items into the customer’s language using the OHI translation mechanism (see the `.ins` script).

Notes on parameters:

- `pi_file_location`: the database directory where the output files are written.
- `pi_module_name`: the name of the module in format SVS<nnnn>R (max. 8 positions).
- `pi_schema_owner`: the (custom) schema owner of the compiled package.
- `pi_table_name`: base table of the selection that generates the output. Within the generated package a default query is created on the base table and the specified items/columns. This query can be extended using any additional joins of tables and/or views.
- `pi_alias`: alias identifying the base table of the selection.
- `pi_label`: functional name of the records to be exported.
- `pi_item_tab`: setup of all report items (columns) with their definitions: within this parameter the `report_item_rec` function defines each column in the CSV report. The columns are included into the default query in the generated package.
- `pi_parameter_tab`: setup of the parameters for the main procedure. Returns the record structure for the definition of a batch parameter. The derived type definition by specifying a table and column prevails over an explicit type definition. Data type is required, however.

Sample call:

```

begin
  sys_gen_pck.export_csv_sources
  ( pi_file_location    => 'OZG_TMP'
  , pi_module_name      => 'SVS0004R'
  , pi_schema_owner     => 'SVS_OWNER'
  , pi_table_name       => 'RBH#RELATIONS_'
  , pi_alias            => 'REL'
  , pi_label            => 'Relations'
  , pi_item_tab         => sys_gen_pck.report_item_tabtype
  (
    sys_gen_pck.report_item_rec
    ( pi_name            => 'REL_NUMBER'
    , pi_table_name     => 'RBH#RELATIONS_'
    , pi_column_name    => 'NUM'

```

```

        , pi_data_type      => 'NUMBER'
        , pi_data_length   => 10
        , pi_data_precision => 0
        , pi_prompt       => 'Relation number'
    )
    , sys_gen_pck.report_item_rec
      ( pi_name           => 'NAME'
        , pi_table_name   => 'RBH#RELATIONS_'
        , pi_column_name  => 'NAME'
        , pi_data_type    => 'VARCHAR2'
        , pi_data_length  => 255
        , pi_prompt       => 'Formatted name'
      )
    , sys_gen_pck.report_item_rec
      ( pi_name           => 'DATE_OF_BIRTH'
        , pi_table_name   => 'RBH#RELATIONS_'
        , pi_column_name  => 'N_DATE_OF_BIRTH'
        , pi_data_type    => 'DATE'
        , pi_prompt       => 'Date of birth'
      )
    , sys_gen_pck.report_item_rec
      ( pi_name           => 'GENDER'
        , pi_table_name   => 'RBH#RELATIONS_'
        , pi_column_name  => 'N_GENDER'
        , pi_data_type    => 'VARCHAR2'
        , pi_data_length  => 1
        , pi_prompt       => 'Gender'
      )
    )
    , pi_parameter_tab => sys_gen_pck.batch_parameter_tabtype
      (
        sys_gen_pck.batch_parameter_rec
        ( pi_name           => 'P_DIRECTORY'
          , pi_data_type    => 'VARCHAR2'
          , pi_data_length  => 30
          , pi_data_precision => 0
          , pi_comment      => 'Database directory
                                where the CSV file is written to'
          )
        , sys_gen_pck.batch_parameter_rec
        ( pi_name           => 'P_SUB_TYPE'
          , pi_table_name   => 'RBH#RELATIONS_'
          , pi_column_name  => 'SUB_TYPE'
          , pi_data_type    => 'VARCHAR2'
          )
        )
    )
);
end;

```

NOTE: the parameter P_DIRECTORY is required. Additional parameters can be added. As a result of what is specified above, the SVS<nnnn>R_PCK will contain:

- parameter validation
- record type for the record to be retrieved
- write the header text with the prompts (translated when applicable)
- write the records one by one into the file using the named columns

The SVS<nnnn>R_CUST_PCK performs the actual data processing and contains a generated query: a select on the base table using all specified columns. The query can be modified/extended but the column aliases must match the definition in the SVS<nnnn>R_PCK.

7.8. Generator: CSV Import Batch

The procedure sys_gen_pck.import_csv_sources creates a template to import and process a CSV file into the OHI Back Office database.

A maximum of twenty columns can be processed. The process_row procedure processes row by row the CSV file.

7.9. Generator: XML Import batch

The procedure `sys_gen_pck.import_xml_sources` creates a template to import and process a XML file. It uses XQuery to process the data stored in an XMLTYPE datatype.

7.10. Other output scripts

It is possible to write your own scripts without using the template generator.

7.11. Module installation script

Once the script is defined in the application it could be necessary to have the same configuration on another environment like the production database.

The definition of a script, including the system messages for this specific script can be exported into an installation file which than can be run against any other OHI Back Office environment to create the same script definition.

The procedure `sys_gen_pck.write_module_ins_file` creates the file, typically with the naming convention `<module code>.ins`.

Sample call:

```
begin
    sys_gen_pck.write_module_ins_file
    ( pi_module => 'SVS0004R'
      , pi_file_location => 'OZG_TMP'
    );
end;
```

The system messages will be included based on the code of the system message. For this the first part of the message code should be equal to the module code.

7.12. Creating output in a specific character set

One of the things a custom batch can do is creating a file, like a CSV file. As the database has the AL32UTF8 character set all output by default will be in this character set.

Each script has the opportunity to create the output in a specific character set. To do this the character set should be set at the script itself and one should use the supplied functions from `ALG_BATCH_PCK`: `create_file`, `write_line` and `close_file` as well as a call to `alg_batch_pck.set_charset`. This last call is not required when using one of the batch templates provided in `SYS_GEN_PCK`.

NB. When using `UTL_FILE` to create or process a file no explicit conversion will be done to another character set and AL32UTF8 is presumed.

8. HTTP Links

HTTP Links allow users of the OHI Back Office GUI Application to view or process related data in an external application. For each HTTP link you can configure a HTTP request template to the target application and the OHI screens which will display the HTTP link.

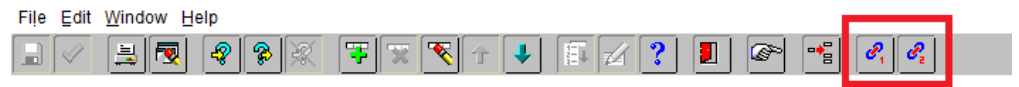


Figure 35: HTTP link buttons

At runtime the HTTP link will open an extra browser window to send a HTTP request to the target application, substituting placeholders with runtime values derived from the current OHI screen.

Note that the target application must be an HTTP application!

Up to nine HTTP links can be defined per screen.

8.1. Configuration

Configuration of an HTTP link in OHI is done using two configuration screens, one for defining the URL and its parameters (HTTP-Link) and one for assigning a link to a menu button for an OHI screen (Module HTTP-Links).

8.1.1. HTTP-Link

The HTTP-Link screen (Menu option: System / Management / General / HTTP-Link) is used to define the URL and an optional tooltip and or icon (see Figure x: HTTP Link definition screen).

To specify a link the following parameters must be supplied:

Parameter	Description	Mandatory
Description	The name of the link. This name will be used in the next screen to identify the link	Yes
Tooltip	A short name for the link that will be shown when the user hovers with the mouse over the button	No
Icon	An alternative icon	No
URL	The URL to link to. When using parameters some syntax rules apply.	Yes
Active	An indicator for activating or deactivating the link.	Yes

If the Active indicator is not checked the link is disabled in any screen where it is used. If the Active indicator is checked it depends on the Active indicator for the particular screen where the link is used whether or not the link is enabled.

When using parameters some extra syntax rules are applicable for the URL. See the description for the next screen below for the correct formatting of the URL.

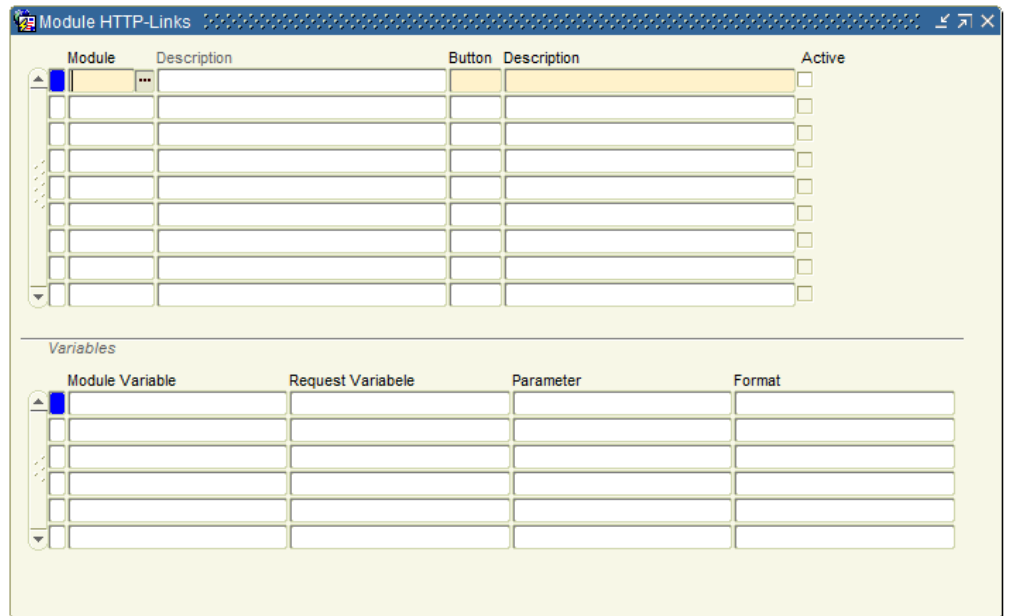


Figure 38: Linking HTTP-links to screens and defining variables

In the upper block of the screen the actual linking between an HTTP-Link and a screen (module) is done. These parameters are described in the table below. There should be no surprises.

Parameter	Description	Mandatory?
Module	The code of the module.	Yes
Button	A number from 1 to ??	Yes
Description	The name of the link as specified in the screen HTTP-Link	Yes
Active	Indicator enabling or disabling the link <i>for the Module</i> .	Yes

The Lower block of the screen is the really interesting part. Here the possible parameters are defined that can be used in the HTTP request when clicking on the button for the link. Every row in the lower block specifies a single parameter for the selected link in the upper block:

Parameter	Description	Mandatory?
Module Variable	The Dutch name of the variable in the OHI screen	Yes
Request Variable	The request variable as it occurs in the URL specified in the screen HTTP-Link	No
Parameter	The request parameter as it appears in the URL	Yes
Format	An optional formatting step before the data is inserted in the URL.	No

8.1.3. Example

Let's use an (admittedly unrealistic) example: suppose we set up an HTTP-Link as in Figure x. It is a link to a well known website and we will use it to search for videos that a relation registered in OHI has posted. Who knows, maybe he has posted videos of him or herself participating in dangerous sports?

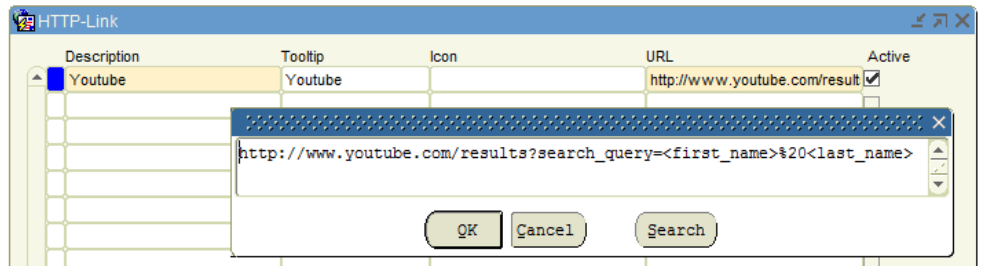


Figure 39: Example HTTP Link

We will then attach the link to the Maintain Relation screen, which is module REL1001F. Also the family name and first name(s) have to be included in the URL. Figure 39 shows how to do it.

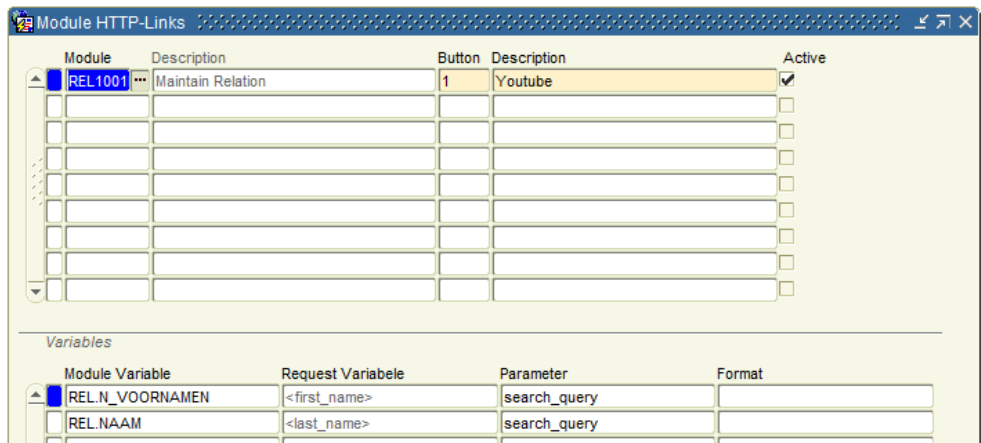


Figure 40: Example set up for a link

Note that when specifying the URL and the request variables the angular brackets (< and >) are not optional.

Now when entering a saving a relation with, say, last name Doe and first name John and clicking on the button that holds our link then the values will be substituted:

`http://www.youtube.com/results?search_query=<first_name>%20<last_name>`

becomes:

`http://www.youtube.com/results?search_query=John%20Doe`

The browser will link to this URL in a new page.

9. OHI Back Office Business Services

The OHI Back Office Business Services add integration facilities for a service oriented environment. The business services are part of a service layer for retrieving and updating core OHI Back Office data.

The 'Find' and 'Get' services are used to retrieve data from OHI Back Office. 'Write' services are used to store data in the OHI Back Office database.

OHI BO Business Services are implemented as PL/SQL services and made available through synchronous SOAP/HTTP web services.

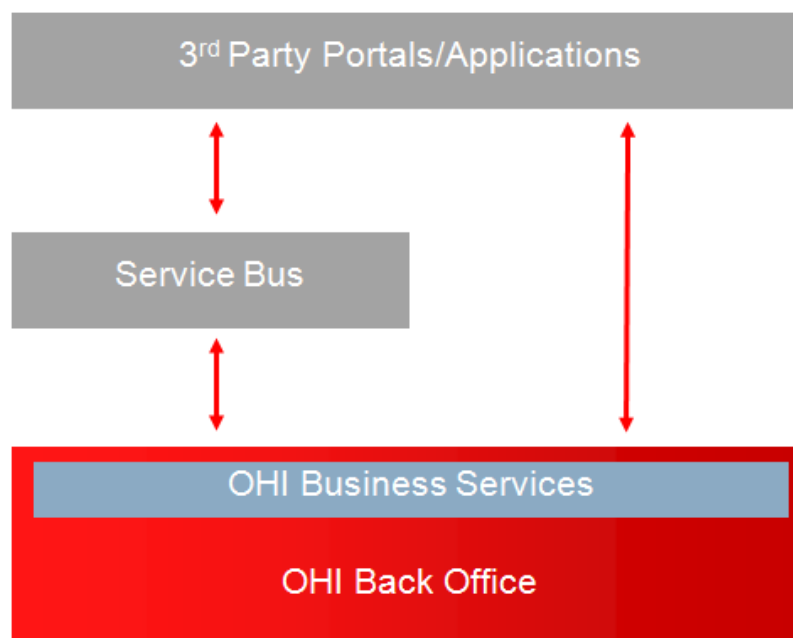


Figure 41: OHI Business Services

9.1. Architecture

The OHI Back Office Business Services share the following characteristics:

- Segmentation into 'Find', 'Get', 'Write' and miscellaneous services to keep the complexity of each service within bounds.
- Each business service is implemented as a web service operation.
- Each web service is based on a WSDL which references versioned XSD files.
- Each web service is implemented as a SOAP 1.1/HTTP, document-style web service.
- Each business service is atomic and stateless ('fire and forget')
- Each 'Write' business service must be idempotent, which means that the response and effect must be the same for different calls with the same data.
- The actual work is carried by a PL/SQL 'service'. The web service acts as an interface between the caller and the PL/SQL service.

A more complete set of characteristics can be found in [Doc\[2\]](#).

9.2. Implementation

The starting point is a WSDL designed by OHI Back Office. The WSDL defines each operation with its inbound and outbound messages. The messages and underlying

XSD types are defined in versioned XSD files. Where appropriate, XSD types use enumerations to translate OHI domain values to meaningful values.

SVL web services consist of the following components:

- A Java web service (WAR file) based on a WSDL designed by OHI Back Office. For each WSDL operation, the Java web service maps the inbound and outbound objects to SQL types which are processed by;
- a PL/SQL web service package which contains the business logic to transform and process the;
- SQL types which hold the content of the inbound and outbound objects.

9.2.1. Difference with C2B architecture

The C2B services were generated from data derived from Designer to call existing PL/SQL packages. Although this 'inside out approach' lent itself well for 100% code generation, the resulting web services were rigid and required application developers to transform objects to the format used internally by OHI and vice versa.

The SVL services are designed 'outside in' which means that every component can be derived to an XSD definition. This allows greater flexibility and contributes to a more user friendly XML content.

9.2.2. The PL/SQL Service

The development of a business service starts with an XSD. Once the XSD for a web service is completed, the SQL type for holding element data are generated as well the Java classes to process XML content to object instances and vice versa.

The PL/SQL part of the web service is handled by a 'PL/SQL service' : a PL/SQL package which implements each business service as a packaged function, using SQL types for processing content.

9.3. Find, Get and Write Services

There are four types of service:

- Find services
These are typically for retrieving a list of objects to select from: many objects, few details.
- Get services
A 'get' service returns a single object with detailed information.
- Write services
Write an object to the OHI Back Office database.
- Miscellaneous services
Examples: a service to get the next free relation number or a service which acts as a 'service consumer' to an external web service.

With these different service types it should be possible to build a client application and keep the complexity of each service within bounds.

9.4. Write Services

The current, second-generation of write services supports idempotent behavior, selective updates and processes time-valid data. The first service of this new breed is the WriteRelation service.

The existing first-generation 'write' services will be migrated to the new paradigm over time.

9.4.1. Idempotent behavior

'Write' services should have idempotent behavior to ensure that each subsequent call to a business service with the same data will return the same response and have the same effect as the first call.

Note that this requirement does not apply for 'Find' and 'Read' services, because the contents of the data base may have changed between subsequent service calls.

9.4.2. Selective updates

'Write' services are used both for inserting and updating data into the OHI Back Office data base. When updating, 'write' services support selective updates to allow the caller to send a partial message. The advantage is that the calling application only needs to know a subset of the data which can be processed by the business message. For example a self-service application only needs to have very little data to allow an end user to update his residential address.

In the example below, the name and phone number are set for relation 1864856800:

```
<v11:Person>
  <v11:relationNumber>1864856800</v11:relationNumber>
  <v11:name>Bakker</v11:name>
  <v11:phoneNumber>06-51227410</v11:phoneNumber>
</v11:Person>
```

If we want to change the name, we can simply pass the new name:

```
<v11:Person>
  <v11:relationNumber>1864856800</v11:relationNumber>
  <v11:name>Slager</v11:name>
</v11:Person>
```

We can also wipe the contents of a column, for example the phone number:

```
<v11:Person>
  <v11:relationNumber>1864856800</v11:relationNumber>
  <v11:phoneNumber></v11:phoneNumber>
</v11:Person>
```

9.4.2.1. 'Zero' update

Leaving out a tag means that its related data is left untouched. This is the cornerstone for selective updates.

9.4.2.2. Use *xsi:nil* to remove existing values

With empty tags you can wipe lists and simple string values. If you want to wipe values which are enumerations or other data types, you should use the nil attribute, like below:

```
<v11:startDate xsi:nil="true"/>
```

Note that the xsi namespace should refer to <http://www.w3.org/2001/XMLSchema-instance>.

9.4.2.3. Lists

Lists can have 0 or more elements which together are enclosed in a list-tag, like in the example below:

```
<v11:Person>
  <v11:relationNumber>1864856800</v11:relationNumber>
```



```

<v11:bankAccountList>
  <v11:bankAccount>
    <v11:accountNumber>
      NL42RABO0111750768
    </v11:accountNumber>
    <v11:bankRelationNumber>
      1525725800
    </v11:bankRelationNumber>
    <v11:bankAccountType>IBANAccount</v11:bankAccountType>
    <v11:countryCode>NL</v11:countryCode>
    <v11:currencyCode>EUR</v11:currencyCode>
  </v11:bankAccount>
</v11:bankAccountList>
</v11:Person>

```

To support selective updates, lists are optional. If you do not want to update a list of details, just omit the list and its surrounding list tag:

```

<v11:Person>
  <v11:relationNumber>1527308300</v11:relationNumber>
  <!--<v11:bankAccountList/> -->
</v11:Person>

```

Likewise , if you want to delete the list, use an empty list tag:

```

<v11:Person>
  <v11:relationNumber>1527308300</v11:relationNumber>
  <v11:bankAccountList/>
</v11:Person>

```



Note: if you add a list to the request you must include all elements. You cannot add a list with a single element just to update the single element. In that case all other elements will be deleted.

9.4.2.4. Time-valid lists

Time-valid lists are used to create and update data with a start and end date, such as addresses, marital statuses etc.

They share some characteristics with ordinary lists:

- Time valid lists are optional: the list related data in the OHI Back Office database are not updated if you omit the list tag altogether.
- All list-related data in the OHI Back Office are deleted if you specify an empty list tag.

The difference is that you can use time-valid lists to update the current situation without removing past data.

Consider the following example to register that Peter is no longer married since 1 January 2013:

```

<v11:maritalStatusList>
  <v11:maritalStatus>
    <v11:startDate>2013-01-01</v11:startDate>
    <v11:maritalStatus>
      dissolved marriage / dissolved registered partnership
    </v11:maritalStatus>
  </v11:maritalStatus>
</v11:maritalStatusList>

```

This information is processed as follows:

- The start date of 1 January 2013 is used as a reference date.
- Peter's previous marital status record (married) is ended by 31 December 2012
- Peter's marital status records starting after the reference date are deleted (if they exist)
- A new marital status record to indicate Peter's current status is created with a start date of 1 January 2013.

9.4.2.5. Segmented time-valid lists

The mechanism described above is too coarse for processing time-valid information like addresses, since you may have different home and postal addresses at any point in time.

This is solved with segmented time-valid lists: this means that the list is processed once for every segment, for example 'address type'.

Consider the following example where John's home address is updated:

```
<v11:addressList>
  <v11:address>
    <v11:startDate>2010-06-04</v11:startDate>
    <v11:addressType>Home</v11:addressType>
    <v11:street>Haverstraat</v11:street>
    <v11:houseNumber>41</v11:houseNumber>
    <v11:postalCode>3511NB</v11:postalCode>
    <v11:countryCode>NL</v11:countryCode>
  </v11:address>
</v11:addressList>
```

This information is processed as follows:

- The start date of 4 June 2010 is used a reference date for John's home addresses
- John's previous home address is ended at 3 June 2010
- John's home addresses starting after the reference date are deleted.
- A new home address is registered starting 4 June 2010
- John's postal addresses remain untouched.

We can update John's home addresses and postal addresses in one go:

```
<v11:addressList>
  <v11:address>
    <v11:startDate>2010-06-04</v11:startDate>
    <v11:addressType>Home</v11:addressType>
    <v11:street>Haverstraat</v11:street>
    <v11:houseNumber>41</v11:houseNumber>
    <v11:postalCode>3511NB</v11:postalCode>
    <v11:countryCode>NL</v11:countryCode>
  </v11:address>
  <v11:address>
    <v11:startDate>2010-07-01</v11:startDate>
    <v11:addressType>Postal</v11:addressType>
    <v11:street>Postbus</v11:street>
    <v11:houseNumber>306</v11:houseNumber>
    <v11:postalCode>3300AH</v11:postalCode>
    <v11:countryCode>NL</v11:countryCode>
  </v11:address>
</v11:addressList>
```

Note that an empty address list will delete all John's addresses:

```
<v11:addressList>  
</v11:addressList>
```



Note: segmentation is not necessarily restricted to a single element (like address type in this case)



Note: consult the functional specification to find out whether a time-valid list is segmented and which elements are used for segmentation.

9.4.2.6. XSD types for Write services.

When examining an XSD for a web service with 'Write' operations you will find that the complex types used by Write Services are prefixed with 'PX'.

You will also find that these complex types largely consist of optional elements. This is needed to support selective updates.

There is a downside to this optionality: if you leave out many elements when entering new data, your inbound XML will still validate correctly against the XSD.

However when sending the request, the OHI Back Office business rules come into play and raise exceptions if your data is incomplete or incorrect.

It would be too complex to document which business rules you may encounter.

Our advice for validating a client application using a 'Write' service would be to always include various tests with new data.

9.5. Error Handling

Two types of SOAP faults when running a web service operation:

- Functional SOAP fault: indicates an error which occurred in the PL/SQL service.
In many cases the functional SOAP fault will contain CDM Rule Frame error messages rather than error messages created specifically for the business services.
- Technical SOAP fault: any other fault except a functional SOAP fault.

10. Service Consumers

Service Consumers are 'clients' for web services (often also named 'business services') that are called by OHI Back Office. These web services are maintained by other parties like VECOZO.

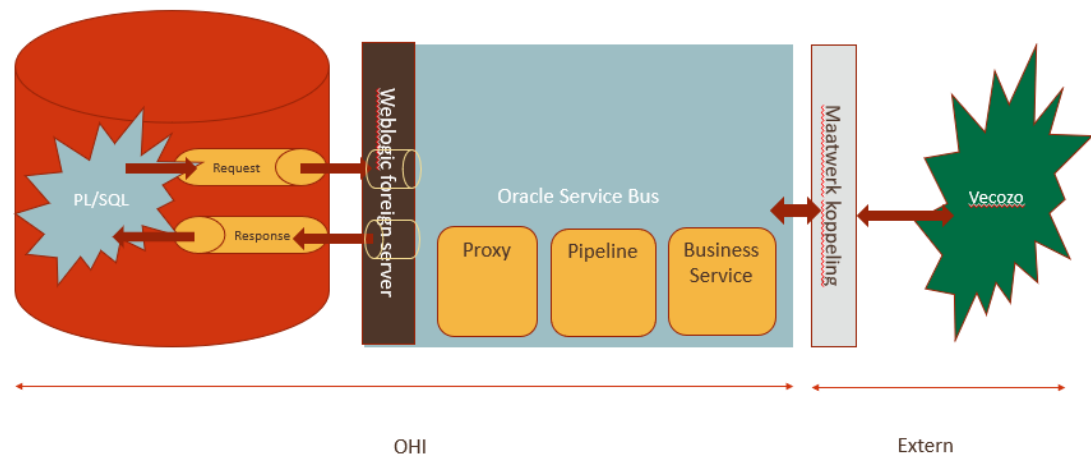
OHI Back Office provides a calling interface to communicate with these services. Part of the solution offered by OHI Back Office is the optional use of an Oracle Service Bus (OSB) project that is provided by OHI.

This chapter deals with situations where you as OHI customer have decided not to use the OSB project definition delivered by OHI. In these situations, a custom solution is needed.

The chapter contains next to that a paragraph describing how to implement custom code calls to the pl/sql packages that implement the web service calls within the OHI Back Office database.

For a high level overview of the Service Consumer solution architecture please consult **Doc[4]**. **That document** describes the background and requirements for the Service Consumer solution. It provides a context for the functionality provided by the OSB project definition that needs to be replaced when the OSB project definition is not used.

The document also provides details about how to access the queues that are needed in any custom solution. A high level architectural overview of the regular implementation using the OSB is shown below.



Although queues are being used to transport the messages, the pl/sql call in the process itself acts as a synchronous process waiting for a response before continuing.

10.1. WSDL transformation

A service consumer is a client implementing the call to a (external) web service. At this moment all external services consumed by OHI are SOAP based web services based on a so called WSDL. That is a file with extension .wsdl that contains the 'contract definition' for the web service. These WSDL files are provided by the owner of the service.

For security and maintenance reasons OHI Back Office provides its own slightly modified version of these WSDL files.

These OHI WSDL files can be found in the \$OZG_BASE/xml folder on the application server of an OHI Back Office environment. The WSDL files can be recognized by the following pattern: SVLxxxxC.wsdl, where xxxx is a number between 0000 and 9999.

In most cases only the target namespaces in the WSDL file have been altered to force at least a proxy between the OHI Back Office call and the outside world. In some cases some more changes have been made to the WSDL in order to use it with the OHI Back Office implementation.

10.2. Request and Response JMS payload queues

When a request from OHI Back Office is issued to an external web service, a JMS (Java Message Service, a standard) structured message with a JMS payload is placed on an Oracle Advanced Queuing (AQ) queue in the OHI database. The name of the queue is ALG_SVC_REQUEST_QUEUE.

In this JMS message a JMS message type is set with the code of the service, which is the same as the name of the WSDL, to be able to identify the type of the message. A correlation id is added as standard JMS property to identify the specific message instance.

After the message is put on the request queue the calling process waits for the response message, with the same correlation id, to appear on another AQ Queue in the database. This response queue has the name ALG_SVC_RESPONSE_QUEUE. This queue also expects a JMS payload.

The response message is recognized by the same correlation id as provided for the request message. If no response for the correlation id is received by the calling process within a given period (the expiration time) the calling process will raise a timeout error.

10.3. Message Processing

The AQ Queues in the database can be exposed in WebLogic as so called foreign server JMS queues. For more detail please see **Doc[4]**. These JMS Queues adhere to the Java Messaging Standard and can be used to enqueue and dequeue messages by any technology that adheres to the JMS standard, such as Java, Service Bus, etc. Part of the Service Consumer solution provided by OHI is an Oracle Service Bus (OSB) project to dequeue the message from the request queue, send it to the external provider, and place the response back as a message on the response queue. As stated this project is optional to use. If the OSB platform is not available for use it is also possible to process these messages with a custom solution that implements the same functionality.

This paragraph assumes the OSB project definition as provided for OHI Back Office is not used and instead a custom solution is used.

There are several ways to get the message from the request queue, such as dequeuing in the database, dequeuing with a JMS client from Weblogic, etc. The message from the queue can then be sent to the service provider, after transforming it from the OHI Back Office WSDL format to the service provider WSDL format. This may be done in the same custom process or maybe left to a separate proxy service. The corresponding response needs to be placed on the response queue.

It is important to provide the same correlation id in the JMS property of the response message that was provided in the request message. This is the only way for OHI to link the correct response message to the request message.

Make sure you adjust/transform the response message from the service provider to the OHI Back Office version of the WSDL before placing the response message on the response queue, it has to be adjusted/transformed from the definition of the service provider to the definition as known by OHI Back Office.



Note: In most, but not all, cases only the namespaces in the message itself need to be modified.



Note: The response message will be validated against the xsd definitions from the OHI Back Office WSDL files when the message is processed. Not complying with these xsd files will result in an error when the response message is dequeued from the response queue and web service call processing will be terminated for this message.

The response with the correct correlation id needs to be enqueued within the configured timeout, otherwise the response will not be processed., because the calling process (an OHI process in the database) will stop waiting for the response message and return an error.

If the response message arrives after the calling process has timed out, the response message will stay on the response queue until the retention time set on the queue, default 24 hours as set by OHI Back Office, unless an expiration period has been set on the response message. This expiration period causes the response message to be placed in the exception queue if the message is not handled by the calling OHI Back Office process within that expiration period.

Database account OHI_JMS_QUEUE_USER can be used to dequeue and enqueue a message from the queues directly or to view the queue contents. The tables implementing these queues (and their corresponding error/exception queues) are described in **Doc[4]**, in the Troubleshooting chapter, paragraph “6.3 Querying Database Advanced Queues”.

In summary:

- A request message for a web service call is placed on database queue ALG_SVC_REQUEST_QUEUE.
- The type of service call for the request is identified by a JMS type and the individual call is identified by the JMS correlation id.
- The request contents is based on the OHI Back Office version of the WSDL which is available in the \$OZG_BASE/xml folder.
- A custom solution needs to
 - immediately dequeue web service request messages from the request queue
 - implement a call to the correct web service (proxy) that transforms the message from the OHI WSDL to the WSDL of the external provider, calls the external provider and transforms the response from the WSDL of the external provider to the OHI WSDL before returning it to the custom solution
 - enqueue the response on the response queue instantaneously.

This all needs to be done in a fraction of a second or a few seconds at the most. By default a maximum processing time of 15 seconds is configured before the calling process fails.

- The corresponding response message must be placed on database queue ALG_SVC_RESPONSE_QUEUE.
- The response message correlation id must be the correlation id from the request message.

- The contents of the response message needs to comply with the OHI Back Office version of the WSDL.
- We strongly advise to specify an expiration time for each response message.
- The database queues can be accessed as regular JMS queues by configuring Foreign Queue definitions in a WebLogic domain. How to do this is described in [Doc\[4\]](#).

10.4. Custom calls on service consumers

The previous paragraphs contain a description how to implement a custom solution for calling the external web services using the request and response queues as communication means.

The majority of the (Dutch localization) service consumers are being called by the processes in OHI Back Office itself. However some can/must be implemented by calling them through dynamic pl/sql code in a policy acceptance check or a policy completion step.

For that purpose a special interface package is available that implements the call to the external web service. These packages are named 'SVC_SVLxxxxC_AQ_PCK' where xxxx stands for the 4-digit number per service. The Service Consumer Installation & Configuration manual contains an overview of the SVLxxxxC service codes and their names.

In [appendix H](#) some code examples are available as reference, to show how such calls can be implemented.

10.4.1. Error handling

When the call fails for some reason, the exceptions are handled in a standard way.

In case a time-out occurs a specific error is raised containing the correlation id of the request and the used value for the time-out.

In other situations the standard OHI error handling is used in which more regular 'functional' errors are distinguished from fatal technical errors (for example when a table cannot extend to store more records).

When a call fails due to some error that occurred during the call and which resulted in SOAP faults it might be helpful to process the SOAP fault that was returned. For that purpose the calling package offers one or more 'get' functions (depending on the WSDL of the specific service definition) through which the SOAP fault structure can be retrieved. As an example, package SVC_SVL1009C_AQ_PCK contains a function GET_FUNCTIONALFAULTTYPE as well as GET_TECHNICALFAULTTYPE.

In the standard error handling it is always checked whether there is a standard SOAP fault present and if so it is signaled by a specific OHI error which is raised and which contains the leading part of the SOAP fault. When stored as event or script message a more complete SOAP fault is stored as context for the message.

11. Custom Development Practices

By giving you an 'open database', OHI Back Office gives you great freedom to develop your own code.

With freedom comes responsibility.

As we explained in the 'Overview' chapter, OHI Back Office is an 'all in one' engine to process claims and policies for health care payers.

A badly written piece of custom code can slow down the core processes for which it is intended or even block the operation of other, unrelated processes.

What follows here is a minimum of tips to prevent these risks.

11.1. Create a custom schema

You should create a custom schema because you are not allowed to connect to the OHI Back Office schema, let alone create your own database objects in the OHI Back Office schema.

11.1.1. How many schema's do you need?

You may have multiple custom applications which interface with OHI Back Office, for example a self-service portal, custom batch processing scripts, and an interface which processes CRM data into OHI Back Office.

Each of these applications may be created by a different team and have a different release schedule.

In that case, a custom schema per application makes it easier to manage these different applications and split the post-installation work and testing efforts whenever a new OHI Back Office release is installed.



Note: For security reasons it is for obvious reasons not allowed to define an OHI officer with the same name as the schema.

11.2. Use an abstraction layer

We cannot stop progress, so OHI Back Office will keep evolving with every new release. This means many smaller or larger changes to the OHI Back Office data model and PL/SQL packages.

Adapting your custom code to the new release should be as little work as possible .

Using an abstraction layer for your custom code helps you to reduce the amount of code and more importantly, the number of references to the OHI Back Office data model and PL/SQL functions.

This is what your abstraction layer should do:

- Make your code DRY
- Encapsulate business logic with packaged procedures.
- Create custom views to query OHI Back Office data.
- Use the OHI Back Office TAPI's for DML operations.
- Call custom packaged functions from Dynamic PLSQL hooks

11.2.1. Make your code DRY

DRY means Don't Repeat Yourself. It is all about replacing duplicate code with your own functions and function calls until you arrive at the minimum amount of code which can still be easily understood and maintained. This has nothing to do with OHI Back Office, it is just good programming practice.

11.2.2. Encapsulate business logic with packaged procedures

The purpose here is to reduce the amount of code which directly depends on OHI Back Office.

If you have large chunks of custom PL/SQL code, split each chunk in 'control' code (the part which evaluates parameters etc.) and functional code (the part which does the processing). Put the processing part in one or more packaged procedures, so that your controlling code does not do much more than call packaged procedures.

Now review your packaged procedures to separate business logic from DML logic (the code which refers directly to OHI Back Office database objects). Put the DML logic in packaged functions.

Caveat: when defining parameters for your custom packaged functions, do not refer to the OHI Back Office tables to keep the dependency of OHI Back Office code to a minimum.

11.2.3. Create custom views

Whenever the OHI Back Office data model changes, it is much less work to revise a small number of views than to repair many individual queries, because views create another abstraction level.

When defining these views it may be attractive to 'stack' views on top of each other and reach an even higher abstraction level. However, this may hurt the performance of your custom code.

As soon as you need to improve the performance of your custom development code, replace 'stacked' views with individual views based directly on the OHI Back Office tables.

11.2.4. Call custom packaged functions from Dynamic PLSQL hooks

Create your own custom packaged procedures to be called by the Dynamic PLSQL hooks in OHI Back Office.

11.3. Define your transactions

The simplest definition of a transaction is 'everything that happens between two commits'.

A better definition (Wikipedia) is 'A database transaction, by definition, must be atomic, consistent, isolated and durable (ACID)'. In order to comply with this definition you should design your transactions as the smallest unit of work can be committed or rolled back.

The notion of the smallest unit of work is important because the longer the duration of a transaction, the more tables and rows are locked which cannot be updated by other processes at the same time.

The notion of a 'unit' of work is important as well: a well-designed transaction ensures that each time the same actions are performed. If a transaction is defined as a limited set of operations it is easier to debug and reduce side-effects. The limited set of operations also makes it easier to predict the duration of the transaction.

General rules:

- Long sessions will keep tables and rows locked and will hinder other processes.
- Do not put transaction control statements in lower-level routines.
- If you anticipate to process hundreds of thousands of rows within a single transaction it may be wise to create multiple transactions.

11.3.1. Transactions and CDM RuleFrame

OHI Back Office uses CDM RuleFrame to enforce business rules, many of which are only evaluated when the transaction is closed or committed.

You may use

- `Api_algemeen_pck.start_api_transactie`
Calls `qms_transaction.open_transaction` to open a CDM Rule Frame transaction.
- `Api_algemeen_pck.einde_api_transactie`
Calls `qms_transaction_mgt.close_transaction` to close the CDM Rule Frame transaction.

Structuring your transaction is important to avoid confusing results when the rules on the Rule Stack are finally evaluated.

More information about CDM RuleFrame in 'Appendix D - What you should know about CDM RuleFrame'.

11.4. Locking

If your code does not explicitly lock database rows, the locks in your session will be implicit locks resulting from updates on individual rows. These locks will be released when you commit or roll back your transaction.

This strategy would be OK if the application does not have long running transactions.

Since OHI Back Office is batch oriented, your code should always take into account that:

- a long running process may have locked the same data that your code tries to update.
- another process may be waiting for you to release the locks you have (implicitly) set in your transaction.
- a worst case scenario may evolve if your code waits to lock a row while other processes are waiting to lock the rows already locked by your code.

The safest advice is to:

- do not use 'lock table' statements
- use 'select for update nowait' cursors if you intend to update data.

If one of the selected rows is already locked by another transaction, the 'select for update nowait' cursor will raise a 'resource_busy' exception.

It is then up to you to process the 'resource_busy' exception. By default your transaction will be rolled back and control will be handed to the calling program.

11.5. Use Named Parameters

When calling an OHI Back Office function or procedure, you can choose to make the call with 'positional' or 'named' parameters.

```
l_rel_no := api_en_relation_details.adjudicate_manually
          ( 1234567800
            , 'D'
            , 'Approved by management'
            );
```

Figure x: call with positional parameters

```
l_rel_no := api_en_relation_details.adjudicate_manually
          ( p_no           => 123467800
            , p_new_status => 'D'
            , p_explanation => 'Approved by management'
            );
```

Figure x: call with parameters

Using named parameters makes your code easier to understand and maintain. It also makes your code less dependent of a specific version of OHI Back Office because as long as no mandatory parameters are added to the OHI Back Office function, it will compile. On the other hand, if Oracle adds mandatory parameters, the compilation will raise a syntax error which tells you which function call failed to compile.

11.6. Profile your code

As a whole, OHI Back Office works best if there are no long running transactions. Your custom code may hinder other processes if it takes too long to run due to badly performing queries.

You should ensure that you uses the database resources efficiently:

- Open SQLDeveloper
- Connect to a non-production OHI database with production-like volumes
- Copy each query to the workspace
- Select 'explain' to explain your query
- Iteratively refine your query

If you don't use SQLDeveloper, use EXPLAIN PLAN (See SQL Reference for details).

11.7. Close open cursors

When possible use cursor-for loops. Apart from requiring less code, cursor-for loops automatically close their cursors. This is important to avoid the 'ora-01000 maximum open cursors exceeded' exception.

If you explicitly open a cursor, for example to fetch a single row, you should close the cursor as soon as possible. If an exception is raised before the cursor is closed, the cursor remains open and must be explicitly closed (test with %ISOPEN).

11.8. Coding Standards

We did not include any naming or formatting conventions because we felt these would confuse the issues discussed. But using coding standards makes your code more maintainable and more secure (providing you also apply secure coding standards).

12. Deprecated Interfacing options

The following interfacing options are still available but will be obsolete in a future release:

- API layer (PL/SQL Functional API)
- API views (if these still exist)

13. Appendix A - Business event framework datamodel

ALG_EVENT_DEFINITIES	
This table holds the event definitions	
NAAM (Name)	This is a logical name for the event type, for example: AZR_MODIFY_PARTY.
OMS (Description)	For example: Export updates to parties to the XYZ system.
DETECTOR	A custom plug-in procedure which is called to detect events for this event definition. Example: azr_mod_prt_pck.detector.
BEGIN_HANDLER	A plug-in procedure that must be called once before processing events for this type, for example to create a .csv file to which all event data will be written.
HANDLER	A plug-in procedure which is called for every event that must be processed. Example: my_event_pck.handler.
END_HANDLER	A plug-in procedure which is called once after all events have been processed, for example to close a .csv file to which all event data were written.
LAATSTE_DETECTIE_DATUM (last_detection_date)	Used by the framework to record the last date when the detection mechanism was used.
STATUS	Updated by the framework to avoid multiple starts of the framework for this event definition. The status can be: 'K' (ready) or 'L' (running).
RUN_NR	Managed by the framework. All events that were detected in a single run are given the same run number for later processing and reporting.
SCHONINGSINTERVAL_VERWERKT (Purge interval processed)	Defines when (successfully) processed events for this definition may be purged. The default interval is 7 days.
SCHONINGSINTERVAL_MISLUKT (Purge interval failed)	Defines when failed events for this definition may be purged. The default interval is 27 days.
IND_ACTIEF (Active indicator)	Indicates whether the event is currently active.

ALG_EVENT_INIT_WIJZIGINGEN	
This table holds the tables which are monitored by the event for triggered events.	
EDE_ID	Foreign key to ALG_EVENT_DEFINITIES.
TAB_ID	Foreign key to ALG_TABELLEN.
IND_INSERT	Indicates whether insert actions on the table should be signaled.
DPS_ID_INSERT	Foreign key to ALG_DYN_PLSQL_DEFINITIES to fine-tune the insert trigger.
IND_UPDATE	Indicates whether update actions on the table should be signaled.
DPS_ID_UPDATE	Foreign key to ALG_DYN_PLSQL_DEFINITIES to fine-tune the update trigger.
IND_DELETE	Indicates whether delete actions on the table should be signaled.
DPS_ID_DELETE	Foreign key to ALG_DYN_PLSQL_DEFINITIES to fine-tune the delete trigger.

ALG_EVENT_INIT_WIJZIGINGEN	
This table holds the tables which are monitored by the event for triggered events.	
IND_ACTIEF (Active indicator)	Indicates whether the event is currently active.

ALG#EVENTS	
This table stores events with storage clause set to Table.	
EDE_ID	Foreign key to the ALG_EVENT_DEFINITIES table for the event definition that signaled this event.
EDE_RUN_NR	Set by the framework to group event occurrences. The highest run number is stored in the event definition.
CODE	Code retrieved by the detection plug-in for use as a key to process the event. In most cases this will be the primary key that can be used to find the data with which the event is to be processed. To be compatible with storing the event to the queue this should be in format table_id##record_id##DML-type.
STATUS	Records the processing status of an event occurrence. Possible values: 'N' (new), 'O' (pending), 'V' (processed), 'M' (failed).
DATUM_ORIGINEEL (original date)	An optional column that can be used to determine the processing order.
CREATIE_MOMENT (creation date)	This standard column is used for processing in the correct order if the DATUM_ORIGINEEL has not been set.

ALG#EVENT_ERRORS	
This table holds errors during the handling of an event	
EDE_ID	Refers to the event definition that detected this event.
TAB_ID	Foreign key to ALG_TABELLEN, source table of the record
RECORD_ID	Refers to the record of the changed record. Together with TAB_ID this uniquely identifies the record in OHI Back Office.
DML_TYPE	What DML action caused the event
EET_ID	Refers to the event in ALG#EVENTS table.
CODE	Code for event processing.
CREATIE_MOMENT (creation date)	Timestamp when the error occurred
FOUTCODE (Error code)	The code of the error occurred.
FOUTMELDING (Error message)	The error message for the error that occurred.

ALG_EDE_PAYLOAD_TP	
The object for storing events to the queue	
EDE_ID	Refers to the event definition that detected this event.
TAB_ID	Foreign key to ALG_TABELLEN, source table of the record
RECORD_ID	Refers to the record of the changed record. Together with TAB_ID this uniquely identifies the record in OHI Back Office.
DML_TYPE	What DML action caused the event

14. Appendix B - Business event interface ALG_EVENT_INTERFACE_PCK

Procedures		Parameters		
Name	Description	Name	Type	Description
Install	Creates an event definition in the database. Also available as function returning alg_event_definities.id%type.	pi_name	alg_event_definities.naam%type	The name of the event definition.
		pi_description	alg_event_definities.oms%type	The description of the event definition.
		pi_event_type	alg_event_definities.type_signalering% type	Indicates how the event is signaled. Allowable values: D for Detected events T for Triggered events
		pi_storage	alg_event_definities.type_opslag%type	Indicates how events are stored. Allowable values: T for Table Q for Queued
		pi_handler	alg_event_definities.handler%type	The (package) procedure for the handler of the event.
		pi_detector	alg_event_definities.detector%type	The (package) procedure for the detector of the event.
		pi_begin_handler	alg_event_definities.begin_handler% type	The (package) procedure for the begin handler of the event.
		pi_end_handler	alg_event_definities.end_handler%type	The (package) procedure for the end handler of the event.
		pi_purge_processed	alg_event_definities.schoningsinterval_verwerkt%type	Determines after how many days successfully processed events can be deleted from the event table.
pi_purge_failed	alg_event_definities.schoningsinterval_mislukt%type	Determines after how many days failed events can be deleted from the event table.		
deinstall	Removes an event definition form the database. When events still exist for this definition an error is given.	pi_name	alg_event_definities.naam%type	The name of the event definition to be removed from the database.
deinstall	Removes an event definition form the database. When events still exist for this definition an error is given.	pi_ede_id	alg_event_definities.id%type	The ID of the event definition to be removed from the database.
purge_all_events	Removes all events for the given event definition from the database. Can be used prior to the deinstall procedure to remove all events.	pi_name	alg_event_definities.naam%type	The name of the event for which all the event occurrences will be removed.
purge_all_events	Removes all events for the given event definition from the database. Can be used prior to the deinstall procedure to remove all events.	pi_ede_id	alg_event_definities.id%type	The ID of the event definition for which all the event occurrences will be removed
reapply_failed_event	Reset events with the status 'Failed' from a	pi_name	alg_event_definities.naam%type	the unique name of the event definition

Procedures		Parameters		
Name	Description	Name	Type	Description
reapply_failed_event	previous run. Reset events with the status 'Failed' from a previous run.	pi_code	alg#events.code%type	the identifying code of the event
		pi_edc_id	alg_event_definities.id%type	the unique identifier of the event definition
add_event	This procedure must be called by the detector of an event to add an occurrence of the event to the event table or queue. When the storage type of the event is set to Table, the event is only created in case there is no existing event with the given code for the event definition with a status N(ew). When the storage type of the event is set to Queue, the event is always placed on the queue.	pi_code	alg#events.code%type	the identifying code of the event
		pi_name	alg_event_definities.naam%type	The name of the event definition.
		pi_code	alg#events.code%type	The identifying code of the event. Must be in format: Table_id##record_id##dml_type. E.g. 1234##876##U.
		pi_date	alg#events.master_date%type	Optional timestamp for ordering event handling.
add_event	This procedure must be called by the detector of an event to add an occurrence of the event to the event table of queue. When the storage type of the event is set to Table, the event is only created in case there is no existing event with the given code for the event definition with a status N(ew). When the storage type of the event is set to Queue, the event is always placed on the queue.	pi_delay	number	Optional; number of seconds to wait before a message can be dequeued after the message is set on the queue
		pi_edc_id	alg_event_definities.id%type	The ID of the event definition.
		pi_code	alg#events.code%type	The identifying code of the event. Must be in format: Table_id##record_id##dml_type. E.g. 1234##876##U.
		pi_date	alg#events.master_date%type	Optional timestamp for ordering event handling.
add_event	This procedure must be called by the detector of an event to add an occurrence of the event to the event table or queue. In case the storage type of the event is set to Table, the event is only created when there is no existing event with the given code for the event definition with a status N(ew). In case the storage type of the event is set to Queue, the event is always placed on the queue.	pi_delay	number	Optional; number of seconds to wait before a message can be dequeued after the message is set on the queue
		pi_edc_payload	alg_edc_payload_tp	The object type containing the data of the event.
type_payload_to_code	Function which converts a storage type Queue payload type to a storage type Table format. Returns alg#events.code in the format table_id##record_id##dml_type. E.g. 1234##876##U	pi_edc_payload	alg_edc_payload_tp	The object type containing the data of the event.
code_payload_to_	Function which converts a storage type Queue	pi_name	alg_event_definities.naam%type	The name of the event definition.

Procedures		Parameters		
Name	Description	Name	Type	Description
type	payload type to a storage type Table format. Returns alg_edc_payload_tp.	pi_code	alg#events.code%type	The identifying code of the event. Must be in format: Table_id##record_id##dml_type. E.g. 1234##876##U.
code_payload_to_type	Function which converts a storage type Queue payload type to a storage type Table format. Returns alg_edc_payload_tp.	pi_edc_id	alg_event_definities.id%type	The id of the event definition.
		pi_code	alg#events.code%type	The identifying code of the event. Must be in format: Table_id##record_id##dml_type. E.g. 1234##876##U.

15. Appendix C - Tracing

The application code of OHI Back Office has been instrumented to trace the PL/SQL code as it is executed. Both the CAPI packages, batch packages and underlying packages of the OHI Back Office Business Services have been instrumented.

By turning on tracing you get a detailed overview to help you analyze the program flow. This is useful if you want to debug or understand a problem.

You can also instrument your custom code with trace calls.

This tracing information is stored in technical tables as described in this Appendix.

15.1. Activation

15.1.1. Tracing a Batch Request

You can activate tracing for a batch request by setting the pop-list "Debug level" to "Profile", "Debug" or "Debug finer", when creating the request in the submit script request screen (screen module SYSS003F). For some batches it is even possible to limit tracing information to specific pl/sql procedure or functions to prevent many thousands of useless traces lines are registered which make it harder to zoom in on the actual wanted information. This limitation is activated by entering one or more packaged routine names, like 'pckx.prcy; pckz.prcw'.

The screenshot shows a web form titled "Batch Request". At the top, there is a "Batch" field with the value "ZRG3009S" and a "Process Claim" field. A "Submit Request" button is located to the right. Below this, the "Batch Request" section contains several input fields and checkboxes:

- Number: 10000000009308
- Printer: (empty)
- Processes: (empty)
- Max. Hours: (empty)
- # to Process: (empty)
- Start Time: (empty)
- Frequency: (empty)
- Optimizer Mode: ALL_ROWS
- Trace:
- Exceed: 5
- Exceeded Trace:
- Debuglevel: Lit
- Programma's: (empty, circled in red)

15.1.2. Tracing a Forms Session

You can trace a form by setting the indicator "Debug" in the screen for user authorizations (screen module SYS1017F):

15.2. How to Access Trace Logs

The log information is written into two tables:

- ALG#TRACE_SESSION: This table contains session information per debug session, like the session id, the user and module code in case of a forms session.
- ALG#TRACE_LOG: This table contains all the data of a trace per session. It also records the PL/SQL object and the procedure stack of the call to the trace package (ALG_TRACE_LOG).

With the OHI Back Office parameter "DebugLines" the maximum amount of trace records per session can be set as a session can create a huge set or records.

Records in ALG#TRACE_SESSION and ALG#TRACE_LOG will be purged automatically after a certain threshold. This threshold can be set with the BackOffice parameter "RetentieTLG".

15.3. Instrumentation of Custom Code

You can instrument your own code using the various procedures of the ALG_TRACE_PCK package:

Procedure	Description
Enable	Activate a debug session
Disable	De-activate the debug session
Enter	Should be called at the start of a function or procedure. Helps following the flow
Leave	Should be called at the end of a function or procedure.
Log	Provide extra logging
Force_write	Force writing of debug information from memory to the table



Note: Avoid the use of disable / enable in low level code.

16. Appendix D – What you should know about CDM RuleFrame

If you develop custom code to work with OHI Back Office you should know a few things about CDM RuleFrame:

- CDM RuleFrame was developed by Oracle Consulting around 1999.
- CDM RuleFrame uses Oracle Designer as its repository.
- OHI Back Office uses CDM RuleFrame to enforce business rules.
- In 2004-2005, the OHI Back Office team revised parts of CDM RuleFrame to solve performance issues.
- CDM RuleFrame has its own concept of a transaction to add functionality to the standard Oracle transaction.
- During the CDM RuleFrame transaction, rules are pushed to the 'Rule Stack': when a DML action is performed on a row in a table, the generated database trigger code calls the appropriate function in the table-specific TAPI package. This function puts those rules which cannot be evaluated immediately on Rule Stack for later evaluation.
- The rules that were pushed on the Rule Stack will be evaluated only when the CDM Rule Frame transaction is closed.
- You can avoid building a 'random' Rule Stack by explicitly defining your CDM Rule Frame transaction.
- You can start a CDM Rule Frame transaction in your custom code by calling `api_algemeen_pck.start_api_transactie`
- You can end a CDM Rule Frame transaction by calling `api_algemeen_pck.einde_api_transactie` or calling `COMMIT`

17. Appendix E – Modification Mechanism for Policies and Relations

OHI Back Office has a ‘modification mechanism’ to allow changes to policies and relations without hindering the core processes like claims processing.

OHI Back Office can be configured to allow direct DML on relations or to use the modification mechanism.

If the mechanism is used, the policy or relation that is to be updated is cloned, together with related child tables, for example addresses, discounts etc.

The cloned rows have negative ID’s which are the negative value of the original (positive) ID. As the clone is being updated, the original rows are untouched, which allows the core processes to continue.

The cloning of a policy can be done with the `API_EN_POLICY_AND_DETAILS.MODIFY` function. The equivalent function for a relation is `API_EN_RELATION_DETAILS.MODIFY`.

The cloned policy or relation can be updated at will. However a delete on an already existing row in the original version cannot be performed. If it is required to remove such a record the column “`MODIFICATION_STATUS`” (“`mutatie_status`”) should be set with the capital letter “`V`” (from the Dutch term ‘`verwijderen`’).

Once the user has completed the updates to the cloned policy or relation, the status of the clone is set to ‘`READY`’. The clone must then be approved manually or by a batch process (`ZRG4021S`).

Once the changed version has been approved, the clone data is merged with the original data and the clone is removed.

To merge the clone with the original policy use the function `API_EN_POLICY_AND_DETAILS.APPLY`.

The set of functions to manipulate a policy in the api `API_EN_POLICY_AND_DETAILS` are:

Function	Description
Modify	Make a copy of a policy
Apply	Merge the modified copy with the original
Remodify	In case of issues when applying the copy go back to the “Pending modification” status
Cancel	Purge the copy policy
Modification_pending	Check if there is already a copy of the policy

18. Appendix F - Dynamic PL/SQL Types

The following dynamic PL/SQL types are available in the given processed. Column bound checks are available on every functional table.

Name	Type	Purpose
Adresveld (Address field)	Check	Validate the data of an address field against a address reference table
Adresveld bepaling ()	Text procedure	Determine address components based on e.g. the postal code in the OHI Back Office screens
Back Office parameter ()	Check	Provide a validation on the values entered in the Back Office parameter values
Betaalactie (Payment)	Text	Provide a payment description used by the XML output for payments
Betalingsverkeerregel (Payment matching detail)	Text procedure	Determine the correct source (payment or collection) for this payment detail.
Commissieberekening (Commission calculation)	Text procedure	Calculate commission
Commissieberekeningdatum (Commission calculation date)	Text procedure	Set the calculation date
Commissieberekeningverzoek (Commission calculation request)	Text procedure	Request a commission calculation
Commissieregelevaluatie (Commission rule evaluation)	Text procedure	Evaluate a commission rule
Declaratieregel (Claimline)	Check	Validate a claimline
Event (Event)	Check	Provide a trigger condition for the triggered events in the Business event framework
GBA-controle (Population register check)	Check	Perform a check for a relation against a population register
Incassoactie (Collection)	Text	Provide a payment description used by the XML output for collections
Kolomgebonden (Column bound)	Check	Perform a custom column bound check
Naamcomponent (Name component)	Check	Validate the data of a name component
Polis (Policy)	Check	Perform a policy acceptance check
Poliscompleteerstap (Complete policy)	Text procedure	Add additional data to a policy in the apply policy process (complete policy)
Restrictie (Restriction)	Check	Can be used to check if an entity flexfield is allowed on a subset of the data for the entity
Standaardantwoord op dekkingsvraag (Default answer to benefit question)	Text	Provide a default answer for a benefit question
XML (XML)	Text	Add custom data on a record when performing an import of a XML file

Name	Type	Purpose
Zorgvoornemenperiode (Pre-authorization period)	Check	

19. Appendix G - Using a JMS payload queue

Starting with OHI BO release 10.19.1.2.0 a first database queue has been made available that can be used to store/publish/enqueue JMS (Java Message Service) messages. The queue is named ALG_JMS_QUEUE. Goal is to offer the possibility to publish JMS messages on this queue when a certain event within OHI Back Office occurs, providing a possibility to receive and process these messages somewhere in an application outside OHI Back Office to take action on the message.

A pl/sql API routine is available to put messages on the queue. For dequeuing messages for usage outside OHI Back Office the database account OHI_JMS_QUEUE_USER should be created, as described in the OHI Back Office installation and configuration manual. The account can be used directly to access the database and dequeue messages or to create a data source that is used in WebLogic to provide a JMS queue to any JMS client.

19.1. Starting points for the standard OHI queue with JMS payload

It has been agreed with the OHI customers to provide a standard configured JMS messaging queue. As there are many options regarding the use of JMS messaging and as future usage requirements are not clear a number of choices is made regarding the queue being offered. Future enhancements may change this or even introduce additional queues. The goal is to offer a best practice queue and not a queue configuration tool as that does not add much added value when compared to a custom queue.

- A queue is offered, not a topic, meaning that only one consumer can consume a message.
Rationale: A queue can have multiple parallel consumer processes meaning processing the messages on it is scalable. For a topic only one consuming process per client can be defined which limits throughput due to the serial processing. It was agreed that when multiple different consumers for a message are needed the message is republished on a separate topic by the OHI customer.
- No specific ordering or priority functionality is supported.
Rationale: As ordering or priority functionality applies to all messages and the foreseen usage is generic and to support different message types this cannot be supported with a generic queue.
- As message type a TextMessage is chosen, as payload body type.
Rationale: There is a need for structured text messages, very probably XML structured, JSON structured or fixed length character structured. Given the flexible way this type can be used this specific message type offers the best foreseen usage options.
- The queue will be persisted in the database, meaning not memory only.
Rationale: When an event occurs within OHI BO the publishing is part of the transaction that commits the event and in this way the event publishing is reliable and robust, the message is not lost when there is an availability failure.
- The standard CorrelationId property will be filled with a standard identification provided by OHI Back Office.
Rationale: This identification can be used to follow the handling of the event when it is further processed. A unique OHI provided sequence nr is used, currently being unique over this and other future provided JMS queues. In a future enhancement a GUID might be used generated by OHI. It is possible to dequeue a message with a specific CorrelationId.

- Several different types of JMS messages can be published on this queue. To differentiate between the message type the standard JMS property JMSType is used.
Rationale: By dequeuing with a condition on this property only a specific message type can be handled by a specific dequeue process.

19.2. Points of attention

As there is no clear expectation regarding the future usage and no clear requirement the following point of attentions apply for this first implementation:

- There is no specific optimization for when there are many yet unprocessed (still to be dequeued) messages on the queue. When there are for example more than 100.000 messages of a certain JMSType not dequeued this will have a negative impact on for example dequeuing a single message with a specific CorrelationId or with a different JMSType than most of the non dequeued messages. This might be enhanced when there is more experience with actual usage. As such it is advised to dequeue all type of messages within a relatively short time after becoming available.
- Message sizes of all types should be kept relatively small to prevent a situation where additional storage and processing overhead of one specific type might delay the processing of other types. A message text payload of more than 4000 characters should in generally be prevented when using this generic queue. If messages become larger it is strongly advised to first investigate the amount of larger messages that is to be processed in a certain period of time an test with realistic loads whether processing times are acceptable.
- Given the very general nature of the queue the queue itself is a compromise as it can service all types of messages but as such it not optimized for any usage. In this way the queue is not meant to handle large amounts of messages for a specific purpose. Please take notice of this when using the queue and beware of the limitations this imposes. The queue cannot be considered to handle all kind of use cases very well, for specific high load or special type of messages it may not be suited and a separate tailored queue definition might be required. OHI does not support situations where the queue is overloaded or misused given the limitations of the generic definition of the queue. As such the OHI customers are responsible for a sensible use. Where sensible use is also dependent on how the processing infrastructure is sized and configured.

19.3. Interface package

For putting JMS messages on the queue a new pl/sql package is offered named ALG_JMS_INTERFACE_PCK. This package offers an enqueue procedure as well as an enqueue function next to a dequeue function.

19.3.1. Enqueue procedure

The interface definition of the procedure is shown below:

```
procedure enqueue
```

```

( pi_queue      in varchar2
, pi_type      in varchar2
, pi_payload   in clob
, pi_expiration in number default sys.dbms_aq.never
, pi_property_list in jms_property_list_tab
);

```

The name of the queue, ALG_JMS_QUEUE, should be provided for the first parameter (this is checked).

For the second parameter a self chosen type identifying string of at the most 100 characters may be specified.

The payload is the actual textual message and may be a CLOB where it is advised to limit the length to 4000 characters as this is less resource intensive.

The expiration parameter specifies the number of seconds until the message is too old to dequeue and expires. When not passed it defaults to the shown constant (actual value -1) meaning the message will not expire and wait endlessly for being dequeued.

The property list definition provides functionality to add user properties to the JMS header. It is a list of varchar2 or number properties using the following definition in the package specification:

```

type jms_property is record ( str_value varchar2(2000)
                             , num_value number --double
                             );
type jms_property_list_tab is table of jms_property
                             index by varchar2(100)
                             ;

```

Beware, when both the str_value and the num_value are filled the num_value is ignored.

The message enqueue action is part of the surrounding transaction so only visible to dequeue processes when the containing transaction is committed.

19.3.2. Enqueue function

The enqueue function has almost the same definition as the enqueue procedure and currently returns additionally the correlation id as number value. This id is now unique over the different JMS payload queues within OHI Back Office (for the service consumer callout functionality a separate different JMS payload queue is used).

19.3.3. Dequeue function

The dequeue function in this package is currently only meant for OHI internal usage and not documented.

19.4. Enqueue example

A simplified example is given for handling the situation that a policy is created and changed to 'Definitive' for the first time while the inflow was caused by an external party that called the registration web service (like for example the 'Independence' channel).

When such a policy has been fully accepted for the first time (it became 'definitive') the policy number and its current state (which might already have been changed as this processing is asynchronous) needs to be returned.

The example below shows a basic Business Event Framework (BEF) 'handler' where it is expected an Independence reference is stored in a specific flex field within OHI Back Office.

The example is basic, meaning that exception and error handling, usage of the correlation id, unexpected situations and preventing duplicate work is not addressed (only a simple 'sunny day' situation is covered).

It does add 2 user properties to the JMS header by means of the property list.

```
procedure handler
( pi_event in alg_edc_payload_tp
) is
  cursor c_pol
  ( v_id in ver_polissen.id%type
  ) is
    select pol.nr
      , pol.status
      , ewe.waarde_char as independence_ref
    from   ver_polissen pol
      ,   ref_eigenschap_waarden ewe
      ,   ref_entiteit_eigenschappen ege
      ,   ref_eigenschappen eig
    where pol.id          = c_pol.v_id
    and   eig.oms         = 'INDEPENDER#'
    and   ege.eig_id      = eig.id
    and   ewe.ege_id      = ege.id
    and   ewe.tab_id      = pol.pol_tab_id
    and   ewe.record_id   = pol.id
      ;

  l_pol_rec c_pol%rowtype;
  l_corr_id number;
  l_payload clob := '';
  l_lst ozg_owner.alg_jms_interface_pck.jms_property_list_tab;
  l_prp_1 ozg_owner.alg_jms_interface_pck.jms_property;
  l_prp_2 ozg_owner.alg_jms_interface_pck.jms_property;
```

```

begin
  open c_pol(v_id => pi_event.record_id);
  fetch c_pol
  into l_pol_rec;
  close c_pol;

  l_payload := l_payload||'<MSG>';
  l_payload := l_payload||'<POL>'||to_char(l_pol_rec.nr)||'</POL>';
  l_payload := l_payload||'<STATUS>'||to_char(l_pol_rec.status)||'</STATUS>';
  l_payload := l_payload||'</MSG>';

  l_prp_1.str_value := l_pol_rec.independer_ref;
  l_lst('INDEPENDER#') := l_prp_1;
  l_prp_2.str_value := to_char(sysdate, 'YYYYMMDDHH24MISS');
  l_lst('ENQ_DATE#') := l_prp_2;

  l_corr_id := ozg_owner.alg_jms_interface_pck.enqueue
  ( pi_queue      => 'ALG_JMS_QUEUE'
  , pi_type       => 'INDEPENDER'
  , pi_payload    => l_payload
  , pi_expiration => 60
  , pi_property_list => l_lst
  );
  commit;
end handler;

```

20. Appendix H – Service consumer example calls

In this appendix examples are shown in which calls to external web services are made through the standard offered service consumer pl/sql routines for this goal.

Within OHI Back Office a limited set of standardized web service consumers (often also name 'clients') is offered for calling web services as provided by Vecozo and some other parties. Most of these services are called by means of standard OHI processes. See also the chapter about [Service Consumers](#) for more information.

However, a few external web services can also be consumed/called by custom code processing triggers and in order to get an idea how this looks like the examples in this appendix are shown.

For a complete set of service consumers offered please consult the OHI Back Office – Service Consumer Installation & Configuration Manual.

The example code in the following paragraphs can be used in a policy acceptance check or a policy completion step. These are based on the service consumers for retrieving residence information (BRP/GBA) and a check on the legal right to be insured (COV).

Goal is also to give a little insight in options how to handle potential errors that may occur as a result of the call.

20.1. Check on COV

In a policy acceptance check the legal right to have a Dutch health insurance can be checked by calling the COV service. The result of this call will be the result of the policy check.

The example below can give you an idea how such a check can be implemented. This example is specific for (depends on) a certain setup and can be optimized but is just here as a reference to give you an easy start.

Please see the online help of OHI Back Office for the exact and correct setup of a policy acceptance check.

The function shown below is typically part of a custom code package (an 'SVS' package) and can be used to call in the dynamic pl/sql code that implements a policy check. The called `get_pol_nr` function is a local function in the same package.

```
function controle_op_verzekeringsrecht
( pi_pol_nr      in      ver_polissen.nr%type
, po_msg_var1   out varchar2
, po_msg_var2   out varchar2
, po_msg_var3   out varchar2
, po_msg_var4   out varchar2
, po_msg_var5   out varchar2
, pi_controle in      varchar2
) return Boolean
is
```

```
-----
-- Date          By          Version  Description
-- 01-01-2016    J.DOE          9.9      COV-check uitvoeren
```

```

-----
l_retval          boolean := true;
l_gevonden_polis boolean;
l_response        svc_sv11010c_tns_controleerresponse_tp;
l_pol_nr ver_polissen.nr%type := get_pol_nr(pi_pol_nr => pi_pol_nr);
begin
po_msg_var1 := null;
po_msg_var2 := null;
po_msg_var3 := null;
po_msg_var4 := null;
po_msg_var5 := null;

-- Loop door nieuwe ZVW deelnames
for r_dee in (select dee.cli_rel_nr
               ,      dee.datum_ingang
               ,      rel.n_datum_geboorte datum_geboorte
               ,      rel.n_sofi_nr bsn
               ,      cco.mer_code
               ,      cco.c_nummer_extern
from           ver_deelnames dee
join          ver_polisproducten plt
on            dee.plt_id = plt.id
join          pas_pakketten pak
on            pak.code = plt.mpr_pce_pak_code
join          ver_pol_collectieve_contracten pcc
on            pcc.pol_nr = plt.pol_nr
and           dee.datum_ingang between pcc.datum_ingang
               and nvl(pcc.datum_einde,dee.datum_ingang)
join          ver_collectieve_contracten cco
on            cco.nr = pcc.cco_nr
join          rbh_relaties rel
on            rel.nr = dee.cli_rel_nr
where         plt.pol_nr = l_pol_nr
and           dee.mutatiestatus = dom_mutatie_status.nieuw$
and           pak.type_verzekering = dom_typen_verzekering.zorgverzekeringswet$
               )
loop
if r_dee.mer_code = 'ALIN1'
and r_dee.c_nummer_extern = 2
then
l_response := sv11010c_impl_pck.controleer_cov
              ( pi_bsn           => r_dee.bsn
                , pi_geboortedatum => r_dee.datum_geboorte
                , pi_peildatum     => r_dee.datum_ingang
              );
l_gevonden_polis := false;
if l_response.m_controleerresult.m_resultaat is not null
and l_response.m_controleerresult.m_zoekresultaten.m_zoekresultaat.count > 0
then

```

```

for l_index in
l_response.m_controleerresult.m_zoekresultaten.m_zoekresultaat.first..
l_response.m_controleerresult.m_zoekresultaten.m_zoekresultaat.last
loop
  if l_response.m_controleerresult.m_zoekresultaten.m_zoekresultaat(l_index).m_resultaat =
    'Gevonden'
  then
    l_gevonden_polis := true;
  end if;
end loop;
else
  --No result
  l_gevonden_polis := false;
end if; -- verzoek akkoord
-- Process result

if l_gevonden_polis - Insured elsewhere
then
  if pi_controle = 'P0030'
  and l_pol_nr > 0
  then
    -- New policy
    l_retval := false;
    po_msg_var1 := r_dee.cli_rel_nr;
    po_msg_var2 := to_char(r_dee.datum_ingang, 'dd-mm-yyyy');
  elsif pi_controle = 'P0031'
  and l_pol_nr < 0
  then
    -- Existing policy, process manually
    l_retval := false;
    po_msg_var1 := r_dee.cli_rel_nr;
    po_msg_var2 := to_char(r_dee.datum_ingang, 'dd-mm-yyyy');
  end if;
end if;
end loop;
return l_retval;
end controle_op_verzekeringsrecht;

```

The call to this service, a policy acceptance check dynamic pl/sql definition may look like:

```

begin
  return svb_owner.svb_dps_pck.controle_op_verzekeringsrecht
  ( pi_pol_nr => :pol_nr
  , po_msg_var1 => po_msg_var1
  , po_msg_var2 => po_msg_var2
  , po_msg_var3 => po_msg_var3
  , po_msg_var4 => po_msg_var4

```



```

    , po_msg_var5 => po_msg_var5
    , pi_controle => 'P0030'
  );
end;

```

The above conforms to the interface definition for policy checks as can be queried within OHI Back Office per dynamic pl/sql usage type. For your convenience, this is the definition as used for this specific check:

Interface definition:

Input:

:pol_nr (number): the number of the policy, will be in #1#

Output:

boolean (value true or false)

po_msg_var1 (varchar2(100)): free format text for message, will be in #2#

po_msg_var2 (varchar2(100)): free format text for message, will be in #3#

po_msg_var3 (varchar2(100)): free format text for message, will be in #4#

po_msg_var4 (varchar2(100)): free format text for message, will be in #5#

po_msg_var5 (varchar2(100)): free format text for message, will be in #6#

20.2. Municipal administration (BRP/GBA)

Besides the check on the fact if a policy member is a dutch resident the GBA service can also be used to obtain additional information like the known address (beware this is only allowed for specific pre-defined health insurance related purposes and make sure you comply with the AVG/GDPR regulations).

This additional information can be used in a policy completion step to register this address in OHI Back Office.

The code below is just an example for reference purposes and needs to be adjusted to your requirements. It is incomplete code contained in an anonymous pl/sql block, offered for your inspiration.

It also contains an example of error handling code where the service consumer specific function is used to retrieve the returned SOAP error information.

```

declare
    l_vraag      svc_svl1006c_tns_gbavvraag_tp := svc_svl1006c_tns_gbavvraag_tp();
    l_antwoord   svc_svl1006c_tns_gbavantwoord_tp := svc_svl1006c_tns_gbavantwoord_tp();
    l_gbavfout   svc_svl1006c_gbavfout_tp := svc_svl1006c_gbavfout_tp();

    l_identificatie svc_svl1006c_identificatie_tp := svc_svl1006c_identificatie_tp();
    l_categorieen   svc_svl1006c_categorieen_tp := svc_svl1006c_categorieen_tp();
    l_categorie     svc_svl1006c_categorie_tp := svc_svl1006c_categorie_tp();
    l_rubrieken     svc_svl1006c_rubrieken_tp := svc_svl1006c_rubrieken_tp();
    l_rubriek       svc_svl1006c_rubriek_tp := svc_svl1006c_rubriek_tp();

    l_bsn         number;

    l_dummy_code   varchar2(2000);
    e_ohi_br_exception exception;
    pragma exception_init (e_ohi_br_exception, -20998);
begin
    -- Use the provided policy nr to get hold of a BSN
    -- (or multiple but you need to adjust the code for that)

    .. - l_bsn is filled

    l_identificatie.m_indicatie      := null;
    l_identificatie.m_interneafnemer := 1234567; -- adapt
    l_identificatie.m_gebruiker      := null;
    l_identificatie.m_profielnaam    := null;
    l_identificatie.m_internkenmerk  := 7654321;

    l_rubriek.m_nummer               := '0120'; --BSN
    l_rubriek.m_naam                 := null;
    l_rubriek.m_waarde               := l_bsn;
    l_rubriek.m_omschrijving         := null;
    l_rubrieken.m_rubriek.extend(1);
    l_rubrieken.m_rubriek(l_rubrieken.m_rubriek.count) := l_rubriek;

    l_categorie.m_nummer             := '01'; --PERSON
    l_categorie.m_naam               := null;
    l_categorie.m_rubrieken          := l_rubrieken;
    l_categorieen.m_categorie.extend(1);
    l_categorieen.m_categorie(l_categorieen.m_categorie.count) := l_categorie;

    l_vraag.m_plaatsindicatieverzoek := 0;
    l_vraag.m_identificatie           := l_identificatie;
    l_vraag.m_categorieen             := l_categorieen;

    svc_svl1006c_aq_pck.stelGbavVraag(pi_request => l_vraag, po_response => l_antwoord );

    --do something useful with the data as returned in the l_antwoord response structure
    ....
exception

```

```

when e_ohi_br_exception
then
  -- error code = -20998
  -- a soap fault might be returned from svc_sv11006c_aq_pck
  -- check if it is a fault from the provider or a generic one
  l_gbavfout := svc_sv11006c_aq_pck.get_gbavfout;
  if l_gbavfout.has_data
  then
    if l_gbavfout.m_foutletter||l_gbavfout.m_foutcode = 'G33'
    then
      -- unknown BSN/BSN not found returned from GBA
      -- do something useful like logging a message
    elsif ..
    then
      ..
    end if;
  else
    -- some other error, we will not handle it, reraise
    raise;
  end if;
end;

```

NB. To get the soap fault type call the specific fault function for this service in the exception handler.