ORACLE
INSURANCE

Oracle Health Insurance Back Office

SOAP Service Layer (SVL) User Manual

Version 1.24

Part number: F50672-01

August 1, 2022

CHANGE HISTORY

Release	Version	Changes
10.12.2.0.0	1.3	• Added technical principles at the consumer webservices
10.13.3.3.0	1.4	• Extended with starting points that apply to the find services with mostly optional arguments.
10.14.1.0.0	1.5	• Included paragraph on 'write' services from 'Custom Development for Oracle Health Insurance Back Office' • Added write services to 'common usage' paragraph.
10.14.2.0.0	1.6	• Clarified the use of nil attributes and empty tags to remove values.
10.14.2.1.0	1.7	• Added paragraph 3.4.1.5
10.14.2.1.0	1.8	• Inserted 3.3.2.6 (termination of a time-valid list beyond a given date).
10.15.3.0.0	1.9	• Updated code pieces and example pl/sql code so it reflects the latest structure and is a working example again (some type definitions were changed in the past 2 years and the example code was not adapted).
10.16.1.0.0	1.10	• No changes
10.16.2.0.0	1.11	• Added some more explanation regarding PX write services and processing of time – valid data.
10.17.1.0.0	1.12	• No changes.
10.17.1.4.0	1.13	• Added note about usage in plsql about package state impact.
10.17.2.0.0	1.14	• No changes.
10.18.1.0.0	1.15	• Revised document title • Revised document references • Revised introduction • Textual changes to 'Generic Principles of provider services'
10.18.2.0.0	1.16	• Republished with new part nr.
10.18.2.1.0	1.17	• Added instructions in the return context message info paragraph for how to activate the returning of the error stack which by default no longer is returned, both not separate and not in the message text of technical ORA-errors.
10.19.1.0.0	1.18	• No changes. Republished with different part nr.
10.19.2.0.0	1.19	• No changes, republished with new part nr.
10.20.1.0.0	1.20	• No changes, republished.
10.20.6.0.0	1.21	• Design principles slighty adapted.
10.21.1.0.0	1.22	• No changes, republished with new part number.
10.22.1.0.0	1.23	• No changes, republished with new part number.
10.22.6.0.0	1.24	• Reworded 3.2 'Find Services – Usage notes' to clarify the usage of wildcards. • Textual changes to 3.3 'Write Services'. Clarified the type of write services available. • Textual changes to 3.7 'Differences between provider plsql and web…' clarifying the availability of the call and return context. • Removed 4.2 'Configuration'. Configuration through OHI Back Office parameters has been rendered obsolete with the changes introduced through M-5778 'Replacement service consumer technology'. • Removed references to a Designer Repository in appendix A.

RELATED DOCUMENTS

A reference in the text (doc[x]) is a reference to another document about a subject that is related to this document. Below is a list of related documents:

- **Doc[1]**: Oracle Health Insurance Back Office SOAP Service Layer (SVL) Installation & Configuration manual (docs.oracle.com)
- **Doc[2]**: Oracle Health Insurance Back Office HTTP Service Layer (HSL) Installation & Configuration Manual (docs.oracle.com)

Contents

1 Introduction

The SOAP Service Layer, also known as SVL services, is an optional component of the OHI Back Office application.

SVL services, also known as 'business services' provide generic object-oriented operations on core OHI Back Office data, with 'find' and 'get' operations to retrieve data and 'write' operations to update/add data. These operations are invoked through SOAP/HTTP

Be aware that OHI Back Office also provides 'Use Case services' (aka HSL services). These REST services provide operations to support typical use cases for Dutch healthcare payers. Examples: requesting a new policy, adding an insured member, changing insured products, changing payment method etc. You will find more about HSL services in Doc[2].

Within the SOAP service layer, there are two types of components:

- **provider web services**
  SOAP/HTTP services built by OHI Back Office to be called ('consumed') by client applications in the surrounding environment.

- **web service consumers**
  Java classes built by OHI Back Office to 'consume' third party SOAP/HTTP services from within the OHI Back Office database.

This document describes the generic technical details of the SOAP service layer. Much of the focus is on invoking SVL provider web services.

Note that installation and configuration of the OHI Back Office SOAP service layer components is described in Doc[1]. That document also contains an architectural overview of SVL provider services and SVL consumer services.

1.1 SVL provider services

SVL provider services are intended to be used in a Service Oriented Architecture (SOA) environment. Note that the provider service operations can also be invoked from PL/SQL.

Since the Java layer providing the SOAP/HTTP interface for SVL provider services is built on top of the PL/SQL implementation, this document does not distinguish between the between the two interfaces unless specifically noted. See Appendix A in this document to retrieve the documentation of each SVL provider service.

1.2 SVL consumer services

The SVL consumer services are used as an integral part of the OHI Back Office application and not documented separately.
The current application code invokes and processes (part of) the functionality provided by third party web services.

2 Generic principles of SVL provider services

This chapter describes a number of generic principles that apply to SVL provider services. Understanding these principles and how the services were developed should provide more insight into the use of the SVL services.

2.1 Design principles

The following design principles were used in the development of the SVL provider services:

1. Terminology, terms and documentation will be in the English language.

   Rationale: the service layer is typically used by developers which are required to know English because most tooling documentation is also only available in English.

2. Terminology, terms, structure and contents of the service definitions will be aligned as much as possible over the different Oracle Health Insurance product lines.

   Rationale: to ease implementing functionality which crosses boundaries of different product lines it helps when the same terms, etc. are used. However, strict (technical) dependencies are prohibited to prevent complicating maintenance dependencies, so differences can occur.

3. SVL provider services can be called using SOAP/HTTP and from PL/SQL

   Rationale: the SOAP/HTTP interface (deployed through WebLogic Server) provides an integration point for other applications in a SOA environment, while the PL/SQL intefrface allows the service operation to be called from custom code in PL/SQL and will execute wth much less overhead.

4. Older OHI Back Office web services such as the 'original' C2B services have been phased out and are replaced with functional identical HSL services.

   Rationale: this will help in offering a uniform and consistent way of implementing services; it will ease management and reduce maintenance efforts that finally will benefit customer investments.

5. SVL services will provide generic functionality instead of a specific solution for a specific problem.

   Rationale: by providing a broader functionality SVL services can be used in a broader context. Note that the 'Use Case Services' or HSL services are targeted towards specific use cases in order to provide a simpler interface.

6. SVL services are designed as application to application operations. The calling application is completely responsible for authentication, authorization and additional calling application specific functionality. As such it should never offer a way to directly call an SVL service. This means the security model of the calling application is completely the responsibility of that calling application.

   Rationale: by offering generic service calls for back-end integration of an application with OHI Back Office a wide variety of applications can be supported. It prevents conflicts or limitations imposed by specific identity or authorization functionality within the calling application.

2.2 Technical principles

The following technical principles are followed and may be of influence for how you realise your code to access the SVL provider web services.

1. SOAP 1.1 is used.

   Rationale: this is by far still the most widely used common standard and supported by almost all web service toolkit implementations.

2. Document style web services are used.

   Rationale: this makes the services widely usable because they are implementation and platform independent.

3. WebLogic Server will be used as the standard application server deployment platform.

   Rationale: this is a highly scalable, reliable and robust application server for deploying Java applications that offers a lot of out of the box functionality.

4. Security functionality will be externalized from the web service implementation unless Oracle standards require differently.

   Rationale: by externalizing authorization, authentication and encryption from the service implementation it is prevented that existing customer functionality conflicts with chosen implementation solutions. This means that customers can leverage their investments for as far as these can interact with a Java based WebLogic deployed application. Normally applying open standards Middleware to support these functionalities will be the way to go.

5. The service calls will be stateless.

   Rationale: to serve easy integration each call is stateless; there is no state to be remembered over more than one service call.

6. A single change call will contain one or more atomic transactions.

   Rationale: service calls that change data do this in one or more atomic transactions, which completely fail or succeed, to prevent inconsistent situations can arise.

7. Optionally a form of optimistic locking can be activated. Default no locking will occur.

   Rationale: because services are stateless and to prevent long outstanding locks blocking other transactions a form of optimistic locking can be activated to control concurrency impact. When at a certain moment data is read and the functionality requires that changes be made under the requirement that the earlier read data is not changed, this optimistic locking can be activated. When the service, which changes the data, detects that the data has been touched by a transaction that committed since the last read moment the change will be rolled back and an error will be returned.
   Of course always a record will be locked explicitly immediately before it is changed, to prevent hanging service calls due to outstanding changes on a record. When the actual lock fails an error message is returned and the transaction is rolled back.

8. SOAP error handling will be used to return functional and technical errors.

   Rationale: this eases integration with development tooling which can leverage functionality based on SOAP faults and makes coding easier and less error prone.

9. Functional faults will support language dependent error messages.

   Rationale: although the services are in English the functional error messages returned will use the multi-language support as present in the OHI Back Office application; this to be able to return language specific messages based on the calling context.

10. Standard Java logging functionality will be offered for error, informative and debug level log messages.

    Rationale: by adhering to a common standard logging mechanism this will be easier to configure and use for system administrators who are experienced with Java based application management.

11. The user manual focuses on a 'standardized approach' for synchronous provider web services.

    Rationale: The business services are currently implemented as such.

12. Additional standardized requirements will be implemented as much as possible through applying standardized technology.

    Rationale: goal is to focus on delivering functionality and be open for most architectural and infrastructural environments; this means that limited development time is not spent on developing proprietary solutions that can be implemented also through standard technology stack software. A number of requirements should be implemented through more or less standardized use of Oracle products. But customers may opt out for other choices. Versioning of services is an example of this.

13. Date and date time values are not expected to have a time zone component in them.

    Rationale: the current application does not support time zones and all date and (date)time values are expected to be expressed in the time zone as used within the database. It may be that time zone handling for (date)time values in the web services are added in a future version where these are always converted to a standard time zone.
    When time zones are passed in values it is expected a service bus or different proxy solution will remove this component in the date and (date)time values.

3 Generic usage aspects of SVL provider services

This chapter focuses on the generic aspects of the SVL provider services.

3.1 Common usage behaviour

For retrieving data several operations may be defined. These operations can be distinguished in 3 types for which certain behaviour rules apply.

- Find services
  The find routines can be used to find a set of occurrences given certain arguments. The arguments in itself do not always need to be existing values and may contain wildcards in a number of cases (see later for more generic principles for find services). If no data matches the criteria no response data is returned and no functional error message is given. Simply the fact that nothing is found should make clear that no data is found.

- Get services
  Routines, which implement 'get' functionality, expect identifiers as inputs that identify exactly one occurrence. If the identifiers do not identify an existing occurrence a functional fault will be returned that no data matches the criteria.

- Write services
  Used for selectively updating a single occurrence, for example a relation or provider contract.

3.2 Find Services – Usage notes

There are two main groups of find services. One set contains some required arguments that are selective and make sure a quite limited set of data is searched for. Some examples are find services for retrieving errors, coverages or flex fields for a given claim line. Or finding claim lines for a specified member.

Another group of services are more generic find services with almost all arguments optional. These are meant to search for data based on quite different arguments. For these services it is important to take notice of the following starting points and usage requirements:

- When you enter a value for an argument, it is used as search condition. It is not possible to search for occurrences where that argument is explicitly empty (not entered, 'null' in database terminology).

- For string conditions, the search is usually implemented as a case insensitive search. Exceptions are selective conditions that can be implemented by an indexed access path being present without special functionality to enable efficient case insensitive searches. An example is postal code, this is case sensitive. Searching on a relation name is not case sensitive because special facilities enable efficient case insensitive searches.

- When an argument is used to identify a related selective condition, for example the 'external code type' for an 'external code', both the identifying argument as well as the selective argument needs to be entered. Otherwise the service will fail with an error. Typically the uppercase value or the passed argument will be used as most of these arguments are by definition stored in uppercase internally (so you do not need to care about case sensitivity).

- When a condition is entered that identifies fact data, for example a country code or an external code type, no wildcard is allowed and existing value needs to be

specified. The argument entered will itself be checked for existence before a search is started. The service call will fail if the value does not exist.

- For string conditions on 'fact data', meaning arguments that do not reference a setup identifying value (like a country code or a name of a flex field), wildcards are allowed. Wildcards are specified using 'Oracle database' wildcards like '_' and '%'. A SQL 'like' will be implemented in such situations. Wildcard usage may result in expensive and/or inefficient queries, e.g. placing the wildcard at the beginning of a string.

- There is no protection against misuse of the services by not specifying any selective arguments. It is expected from the implementing party that at least a selective argument is filled with a selective value. This means that for example searching for all relations with an address on number 24 or with a name containing somewhere 'ee'(using argument '%ee%') will result in expensive inefficient queries that on environments with quite some data may result in a time-out of the service call while the query keeps on executing on the database. It is the responsibility of the calling party to prevent such misuse.

- Given the previous bullet it is expected that realistic performance and usage tests are executed on a representative environment (with production like amounts of data and similar resources and load). This is imperative to prevent response time and load problems are only detected in production.

- As a bold measure to prevent very long running services queries it is possible to activate resource limits on the database for the database account that implements the service calls.

  A very simple example to prevent a call uses more than 50.000 block gets is shown below:

  ```
  create profile svl_prof_max_gets limit
  logical_reads_per_call 50000;
  alter user <svl_user> profile svl_prof_max_gets;
  alter system set resource_limit=true;
  ```

  The limit will be applied to newly logged on database sessions. ORA-02395 will be raised when the limit is exceeded. It is a bold way to prevent a connection pool will become exhausted.

3.3 Write Services

The current, second-generation of write services supports idempotent behavior, selective changes and processes time-valid data, e.g. the WriteRelation service.

These services have 'PX' as part of the complex types' names. 'PX' is short for 'PiXels' as their behavior stands for passing a (small portion of a) photo of the situation as it should be after processing the request. The service will compare the existing situation in the database with the situation as it should be and determines what data to add, update or delete to accomplish the data in the database reflects the requested situation.

The paragraphs below describe this in more detail, especially how to pass only a portion of the requested situation, to accomplish selective changes.

3.3.1 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 database may have changed between subsequent service calls.

3.3.2 Selective updates

'Write' services are used both for inserting and updating data into the OHI Back Office database. 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>
```

3.3.2.1 'Zero' update

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

3.3.2.2 Removing values

You can remove values as follows:

- With an empty tag if the element is a list, or string value.
- With the nil="true" attribute if the element refers to a date, enumeration or numerical value.

Example of empty tag for a string element:

```
<v11:phoneNumber></v11:phoneNumber>
```

Example of nil attribute for a date element:

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

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

3.3.2.3 Lists

Lists (which are not time-valid as opposed to time-valid lists discussed afterwards) 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.

3.3.2.4 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. Typically this is data that changes over time and is only valid for a specified period (although the period may be unknown).

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 situation from a certain start date without removing and updating data before that start date.

When you specify time-valid data from a certain start date you are required to specify the complete timeline for the data from that moment as the timeline of data that is specified will completely overrule the existing data from the (oldest specified) start date.

This means you can leave a situation before the oldest specified start date as provided in the time-valid list untouched but data after that date is considered to be the situation that should exist after processing the 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:endDate>2015-12-31</v11:endDate>
    <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 since August 22nd, 2002, with originally an empty end date) is ended by 31 December 2012.
- All 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 January 1st, 2013 and an end date of December 31st, 2015. The situation after that date is not present (is in fact unknown).

3.3.2.5 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. It would not offer the functionality to change only the change of a postal address over time.

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

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 June 4th, 2010 is used as reference date for John's home address (es) from that moment.
- John's previous home address is ended at June 3rd, 2010.
- All John's home addresses starting after the reference date are deleted.
- A new home address is registered starting June 4th, 2010.
- John's postal and other type of 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.

3.3.2.6 Terminating a time-valid list after a given date.

You may want to terminate a time-valid list after a given date. In that case you should enter a time-valid list with a single element, for which you specify an end date BEFORE the start date (the actual value of the end date is not relevant, but it should be before the start date).

Example with non-segmented list:

```
<v11:maritalStatusList>
  <v11:maritalStatus>
    <v11:startDate>2013-01-01</v11:startDate>
    <v11:endDate>2012-01-01</v11:endDate>
  </v11:maritalStatus>
</v11:maritalStatusList>
```

Existing marital statuses overlapping with 2013-01-01 will be terminated at 2012-31-12. Marital statuses starting after 2013-01-01 will be deleted.

If you want to terminate a segmented list you should add the required segment values. The example below demonstrates how to terminate home addresses after a given date. Other address types are not affected.

Example with segmented list:

```
<v11:addressList>
  <v11:address>
    <v11:startDate>2010-06-04</v11:startDate>
    <v11:endDate>2010-06-03</v11:endDate>
    <v11:addressType>Home</v11:addressType>
  </v11:address>
</v11:addressList>
```

Existing home addresses overlapping with 2010-06-04 will be terminated at 2010-06-03. Home addresses starting on or after 2010-06-04 will be deleted.

3.3.2.7 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.

3.4 Call standards

For each service a calling context and a returning context must be provided to call a service routine. The calling context specifies behaviour and the returning context provides feedback about the call.

These two 'contexts' are described by referencing the plsql definitions for these contexts (these are 'published' identically in the Java layer).

For all service calls there are 2 standardized SVL object types which need to be passed as 2 separate parameters to each service call.

- ✓ SVL_CALL_CONTEXT_TP - input parameter set to pass call data
- ✓ SVL_RETURN_CONTEXT_TP - output parameter set to pass return data

These are each described separately but they are related to each other when optimistic locking functionality should be implemented. In this latter situation part of the return context of a preceding call is input for the next call context.

3.4.1 Call Context

The (plsql) definition of the call context is as follows:

```
create or replace force type svl_call_context_tp
as object
( usercontext                  svl_string30_tp
, enforceconsistentread       svl_string03_tp
, enforceunchangedsincescn    svl_int_tp
, sourceidentificationcode    svl_string10_tp
, constructor function        svl_call_context_tp
  return self as result
)
```

The call context is used to define the behaviour of the called service. A number of settings can be provided to the call.

3.4.1.1 Usercontext

The usercontext setting is used to pass the OHI Back Office known and active username which should identify the user that executes the action.

3.4.1.2 Enforceconsistentread

The setting enforceconsistentread is used to ensure that in a service call all eligible data retrieved in and returned by that call is consistent, in the sense that it is all not changed (even if actually committed or it may not be committed) since the call started (so a committed change on the retrieved data by another session may not be visible or have no effect, since the retrieve operation started, to prevent sequential selects in the retrieve call retrieve an inconsistent situation; with other words, the data returned by the call is as it was at the moment of that call, it is a 'stable photo'). When data is retrieved that is changed since the operation started and no technical alternative for retrieving consistent data an error will be raised by the service routine.

For each service it is defined whether the consistent read option is supported or not. If not the service fails when it is asked to implement a consistent read.

IMPORTANT: When a service enforces a consistent read it only offers it correct when in the database the ROWDEPENDENCIES setting is activated for the OHI tables involved. This is a one time only database table reorganization action but will imply a large downtime to implement this for all tables.

3.4.1.3 Enforceunchangedsincescn

The setting is used typically in a change scenario to implement an optimistic locking algorithm. It should contain a numeric value which identifies an SCN value (System Change Number, a sequential change number assigned to changes and also to each committed transaction; please see the standard Oracle database documentation for more information).

If specified a non null value (and the called service supports it) the service should check for all records that will be changed (and perhaps in special cases also for other records which are retrieved in the operation, but this is service specific), whether they are not changed since the 'SCN moment' which is passed (a parameter value is passed for this). The SCN for the retrieved records may not be younger (larger) than the provided SCN (the third parameter specifies this).

Of course for updates/deletes an explicit lock with nowait is always implemented for the records that are affected (immediately before they are changed), disregarding this functionality. This to prevent the service 'hangs' on an outstanding lock. Internally the service determines the SCN for a record as part of the 'select for update' or after that select has succeeded (because the record is locked from that moment on, until the transaction ends or fails and rollbacks).

When a value zero is passed the 'SCN moment' at the start of the service call will be used (this is a special situation).

IMPORTANT: The same remark as in the previous paragraph regarding the activation of ROWDEPENDENCIES applies for a correct working of this functionality.

3.4.1.4 Combining these settings

By combining the previous 2 settings it can be specified for a service call to check whether data is read consistent during a retrieve of this data and to check whether the data to be changed during a (slightly) later service call is not changed in the meantime. This is done by remembering the SCN call moment of the retrieve call

(which is returned in the returning context which is described later) and passing it on to the change call.

For retrieve only functionality the last setting is normally not relevant. Typically only the consistent read option will be supported but their may be exceptions.

For service calls that change data it can be useful to specify both the enforce_consistent_read and the enforce_unchanged_since_scn settings. When both options are supported this means the routine will support a consistent read for 'supporting data' that is retrieved, but for the data which will be changed the enforce_unchanged_since_scn value that is passed, is used to check whether it has not changed since that moment.

3.4.1.5 sourceIdentificationCode

The sourceIdentificationCode setting is used to pass the code which identifies the system from which the action was executed. It should contain a code that is maintained in OHI Back Office window 'System Source' (SYS1145F).

3.4.2 Return Context

The return context looks like shown below.

```
create or replace force type svl_return_context_tp
as object
(
  messageinfo           svl_message_info_tp
, scncallmoment         svl_int_tp
, constructor function svl_return_context_tp
  return self as result
)
```

It typically can contain a list of regular error messages that occurred during the call. These are passed using the messageinfo structure, the first attribute.

Next to the message info also the database SCN number (in fact a 'moment in time identifier' of the last change in the system at a specific moment) of the moment the service call started is returned, if consistent_read is enforced (!) and supported. This can be used in subsequent calls when the calling environment wants to make sure that data which is accessed/changed in the later call is not changed since the call which returned the SCN (see previous paragraphs).

With using the combination of the return context and the call context in subsequent calls an optimistic locking approach can be implemented in interface applications that require this. This is especially useful because of the stateless behaviour of the service calls, it is not possible to use a locking strategy with actual (row) locks.

3.4.2.1 Message info

The definition of the message info is shown below.

```
create or replace force type svl_message_info_tp
as object
( messages              svl_message_lst_tp
, topseverity           svl_string01_tp
, errorstack            svl_string3200_tp
, constructor function svl_message_info_tp
  return self as result
)
```

The first attribute in this message info object contains the actual list of messages. When there is a list of messages present (one or more messages in the list) this means a (functional) error has occurred which is handled in the exception handler of the called service routine. The message info variable contains in such a situation also the most severe level of the messages in the message list (top_severity). When for

example 2 informational messages and one error message are in the list the top_severity is 'Error' ('E').

In the situation that an exception has occurred which is handled and put in the message list also the plsql error stack can be passed through the 'errorstack' attribute. This can help in detecting the cause of programming or application problems.

Beware: normally the errorstack is never passed as this may return unwanted detailed info regarding the internal code setup of the application which may be used by a malicious user who is looking for opportunities to attack and misuse the application. For that reason the errorstack is only returned when debugging is activated (which requires pl/sql access in the same database session) or by specifying the value TRUE for property BOOLEAN_SHOW_ERRORSTACK in the special database wide (global accessible) application context named SVL_GLOBAL_CTX. By executing the command below this memory setting can implemented (meaning it is never persisted and will not unintendedly be copied into a clone of the environment):

```
begin
  alg_context_pck.set_context
  ( pi_naam       => 'BOOLEAN_SHOW_ERRORSTACK'
  , pi_waarde     =>'TRUE'
  , pi_namespace => 'SVL_GLOBAL_CTX
  );
end;
```

Setting it to any other value disables the returning of the error stack contents.

A message on the message list is structured as shown below:

```
create or replace type svl_message_tp
as object
  ( messagecode              svl_string32_tp
  , severity                 svl_string01_tp
  , severitydescription     svl_string100_tp
  , messagetext              svl_string2000_tp
  , helptext                 svl_string2000_tp
  , languagecode             svl_string05_tp
  , constructor function    svl_message_tp
    return self as result
  )
```

This shows a code is passed that uniquely identifies the specific message type (of course more of the same messages can be in the list) next to the severity, the description of the severity, the actual text of the message (in the language as defined by the preference of the user with which identity the service is called, which language is passed back in the lang_code variable) and optionally a help text which applies to that message.

When technical errors occur (typically ORA- errors) the messagetext contains the description of the ORA- error. During regular use this will only be the ORA- error. However, it is possible to receive additional errorstack information, when present, by running in debug mode or by explicitly enabling this for the used database through an application context message, likewise this is done for the errorstack attribute mentioned before, exactly the same instructions apply.

Typically this may be required when such an error occurs due to for example an OHI coding bug. Asking a DBA to activate the setting and re-executing the code may reveal additional information in the messagetext,

3.5 Error and exception handling

When a call fails it can fail gracefully or by throwing an exception. When it fails gracefully a list of messages is returned in the returning context.

In case of an ungraceful failure, which is not handled by an exception handler in the service, an exception is thrown. Within the database this is typically an Oracle exception.

At the web services side SOAP fault messages reflect both situations. The XSD below specifies for this purpose a functional and a technical fault.

```
  <xsd:element name="functionalFaultType">
  <xsd:annotation>
   <xsd:documentation>SOAP Fault which is returned when a functional regularly
handled error has occurred.</xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element minOccurs="0" maxOccurs="unbounded" name="messages"
              type="faultMessageType"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
 <xsd:element name="technicalFaultType">
  <xsd:annotation>
   <xsd:documentation> SOAP Fault which is returned when a technical unhandled error
has occurred. </xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="code" type="xsd:string" minOccurs="0"/>
    <xsd:element name="message" type="xsd:string"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
 <xsd:complexType name="faultMessageType">
  <xsd:sequence>
   <xsd:element name="severityText" type="xsd:string"/>
   <xsd:element name="severityCode" type="xsd:string"/>
   <xsd:element name="messageText" type="xsd:string"/>
   <xsd:element name="messageCode" type="xsd:string"/>
   <xsd:element name="helpText" type="xsd:string" minOccurs="0"/>
  </xsd:sequence>
 </xsd:complexType>
```

All application raised messages or handled exceptions are returned as functional faults and can be handled as such. As long as functional faults are returned it is clear the service call itself is still being executed along all the layers in the technology stack but somewhere during processing in the application functional errors or a handled exception occurs.

When an unhandled exception occurs this is typically returned as a technical fault and can be handled differently. This is more severe and can have all kind of causes. For example the database can be unreachable or down. But it cannot be said by definition that processing should be aborted, that is dependent on the cause of the technical fault. It might well be that the next call succeeds again because there only was a switch over from for example a failing network component to a replacement.

Beware that the structure of the XML messages, both the request as well as the response message, will be validated against the WSDL/XSD definitions. Errors will be returned when the message does not comply with the definition.

3.6 Transaction handling

For web services that can change data (so which implement more than only 'retrieve' operations) the change operation will commit automatically or fail in a consistent way (rolling back partially executed transactions). This behaviour is implemented automatically in the web service implementation.

When using the plsql implementation the calling code is responsible for committing or rolling back partially executed operations. When exceptions are handled in an exception handler be sure a rollback is executed in the exception handler to prevent partially executed transactions are committed (although this will never result in inconsistent data, that is guarded by the business rule implementation layer in the database), resulting in potentially unwanted changes.

3.7 Differences between provider plsql and web service 'service layer implementation'

The plsql and web service 'services' are very similar. In fact the plsql services are 'wrapped' into a web service. Typically there is one database plsql package supporting the operations of the corresponding web service.

In the service layer implementation the supporting web service plsql packages are named SVL_WS_<service> and contain the operations as packaged procedures. A standard function is_alive is offered that is wrapped as web service to check the complete web service technology stack is working fine. The is_alive function returns the version number of the associated package.

The operations, implemented as packaged procedures, accept as input and output parameters of user defined object types that are similarly formatted as the input and response messages in the corresponding WSDL/XSD definitions.

It is not possible to access the call context and the return context in the web service implementation, as they are not present in the WSDL definition.

There is one exception: the calling username can and must be specified in the Back Office properties file for each web service. Please read the installation and configuration manual for how to specify this calling user name. It will be passed as the user context part for the call context as described earlier.

So to state it more clearly, in the plsql implementation it is possible to specify and access the call and return context directly where this is not possible in the web service implementation.

3.8 Example plsql usage scenario

When the plsql implementation is used there are lots of possibilities in how to use the services. They can be combined with SQL and plsql to retrieve and change data directly or they can be used solely.

You need to use a database account to call the plsql implementation and this may not be the OHI object owner.

You can create a specific custom code owner account for this and connect to it. In the example below the account SVL_TEST is created and the SVL objects as well as other OHI objects are granted to the account (although this is not strictly necessary, a role could also be used):

```
spool grants.txt
begin
  alg_security_pck.svl_grants(pi_grantee=>'SVL_TEST');
  alg_security_pck.direct_grants
  (p_grantee=>'SVL_TEST',p_grant_option=>null);
```

```
end;
/

spool off
```

Below a very simple example is given how to retrieve some policy data using a plsql program:

```
declare
  l_call_context    svl_call_context_tp :=
    svl_call_context_tp(svl_string30_tp('MANAGER'),null,null,svl_string10_tp(null));
  l_return_context   svl_return_context_tp;
  l_pol_details_tp   svl_policyanddetails_tp  := svl_policyanddetails_tp();
  l_response        svl_getpoldetbypolnumintres_tp;
begin
  -- activate tracing in a minimal way to make sure serveroutput produces output
  alg_trace_pck.enable;
  alg_trace_pck.set_loglevel(dom_loglevel.profile);
  -- get policy details for a specific policy nr
  svl_ws_ohipolicy_pck.getpolicydetailsbypolicynumberinternal
  ( pi_request       => svl_getpoldetbypolnumintreq_tp
                        (svl_int08_tp(10000042),svl_ohi_date_tp(date '2015-01-01'))
  , pi_call_context   => l_call_context
  , po_response      => l_response
  , po_return_context => l_return_context
  );
  -- get some output from the response
  if l_return_context.messageinfo.messages.count > 0
  then
    dbms_output.put_line(l_return_context.messageinfo.errorstack.string3200);
    for i in 1..l_return_context.messageinfo.messages.count loop
      dbms_output.put_line
      ('Msg('||i||') '||l_return_context.messageinfo.messages(i).messagecode.string32||
       ': '||l_return_context.messageinfo.messages(i).messagetext.string2000);
    end loop;
  else
    dbms_output.put_line('No errors occurred.');
    for i in 1..l_response.policyanddetails.svl_member.count
    loop
      for j in 1..l_response.policyanddetails.svl_member(i).membership.count
      loop
        dbms_output.put_line
        ('Mmb('||i||'-'||j||'): since=<'||
         l_response.policyanddetails.svl_member(i).membership(j).startdate.ohi_date||
         '> name=<'||l_response.policyanddetails.svl_member(i).internalformattedname.string255||
         '> product=<'||
     l_response.policyanddetails.svl_member(i).membership(j).policyproduct.product.productname.string255||
         '> sofi=<'||l_response.policyanddetails.svl_member(i).socialsecuritynumber.string20||'>'
        );
      end loop;
    end loop;
  end if;
end;
```

As you can see the service operation SVL_WS_OHIPOLICY_PCK.GETPOLICYDETAILSBYPOLICYNUMBERINTERNAL is used. As calling user the known application username MANAGER is used. This code is executed using a database account created for this purpose (as using the application object owner directly is not supported).

Beware: The example code contains a call to alg_trace_pck.enable which activates producing trace records in table ALG#TRACE_LOG. The call is needed in this example for serveroutput (calls to dbms_output.put_line) to produce results.
In ‘production’ code tracing should not be enabled. This means that each SVL call resets the package state of the session and as such prevents the use of dbms_output as the upfront activation is overruled.
It is important to be aware of this when writing code that uses package state, that will not work in combination with multiple svl calls. So do not rely on the contents of package variables but use local variables in your code.

3.9 Example provider web service usage scenario

In the situation of a web service of course the WSDL URL should be used to access the WSDL. The system administrators who deploy the web services should provide this URL.

A typical WSDL URL could be:

http://<servername>:<port>/OHIBOWebservices/OhiPolicyService?wsdl

To test whether a service is technically working a tool like soapUI can be used which creates requests for the different operations.

The isAlive operation can be used for the technical test.

However, a simple example, which is similar to the plsql example in the previous paragraph, can also be used:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:v1="http://www.oracle.com/insurance/ohibo/policy/policymessages/v1">
   <soapenv:Header/>
   <soapenv:Body>
      <v1:getCurrentPolicyDetailsByPolicyNumberInternalRequestType
policyNumberInternal="17553"/>
   </soapenv:Body>
</soapenv:Envelope>
```

This will return a SOAP message containing the contents of the returned policy message structure or a SOAP fault message structure in case of problems.

4 Consumer web services

Starting with release 2012.01 a first implementation is offered of consumer services. These services support existing batch functionality that consumes services in the outside world. Where outside world is defined as 'outside of the OHI application'.

It is expected that these services are not directly exposed to the outside world. An intermediate 'gateway' (typically a proxy or a service bus) should implement the call to the real outside world. Through the use of such an intermediate standardized security solutions can be used to protect the communication to the outside world, independent of the OHI Back Office application.

This chapter focuses on implementation aspects for the consumer services.

4.1 Technical principles

The following technical principles are followed and may influence how you realise your code to access the Service Layer consumer web services.

1. SOAP 1.1 is used.

   Rationale: this is by far still the most widely used common standard and supported by almost all web service toolkit implementations.

2. The OHI provided WSDL's are currently look-a-likes for the outside world WSDL's that are offered by VECOZO.

4.2 Gateway setup

Internally a proxy or a Service Bus should be setup to provide the provider web services functionality that can be consumed by the consumer web services.

To know the functionality that should be implemented a WSDL is delivered per consumer web service that describes the interface to be offered.

These OHI provided WSDL's are currently look-a-likes for the outside world WSDL's that are offered by Vecozo. It should be quite easy to identify how to map the data of the external and the 'OHI' WSDL.

For more information, consult the Functional Specification of theme M-2647.

4.3 Error and exception handling

When a consumer web service call fails it fails by throwing an exception. Because the calls are implemented inside the database this is typically an Oracle exception with the error code ORA-29532. These errors are stored as messages that occurred during the batch that executed the consumer services.

The ORA-29532 error code indicates a java exception; the actual error is shown in the error message that follows after the error code.

Below is a non-exhaustive list of possible errors that can occur when a consumer service is called by the OHI Back Office application:

Error	Cause
ORA-29532: Java call terminated by uncaught Java exception: java.rmi.RemoteException: java.rmi.RemoteException:; nested exception is: HTTP transport error: javax.xml.soap.SOAPException: java.security.PrivilegedActionException: **javax.xml.soap.SOAPException: Message send failed: Connection refused**	No web server available at the given location
ORA-29532: Java call terminated by uncaught Java exception: java.rmi.RemoteException: java.rmi.RemoteException:; nested exception is: HTTP transport error: javax.xml.soap.SOAPException: java.security.PrivilegedActionException: **oracle.j2ee.ws.saaj.ContentTypeException: Not a valid SOAP Content-Type: text/html; charset=iso-8859-1**	A web server is available at the given location, but does not accept SOAP messages or cannot respond to the requested message (unknown request)
ORA-29532: Java call terminated by uncaught Java exception: java.rmi.RemoteException: **oracle.j2ee.ws.common.encoding.DeserializationException:deserialization error: java.lang.IllegalArgumentException**	Unknown or invalid (response) message: invalid value
ORA-29532: Java call terminated by uncaught Java exception: java.rmi.RemoteException: **java.lang.NullPointerException:null**	Unknown or invalid (response) message: invalid namespace
ORA-29532: Java call terminated by uncaught Java exception: java.rmi.RemoteException: **java.rmi.RemoteException:Error parsing envelope: (1, 1) Start of root element expected.; nested exception is: javax.xml.soap.SOAPException: Error parsing envelope: (1, 1) Start of root element expected.**	Unknown or invalid (response) message: empty message
ORA-29532: Java call terminated by uncaught Java exception: java.rmi.RemoteException: **oracle.j2ee.ws.common.encoding.DeserializationException:unexpected element name: expected={urn:http://www.oracle.com/insurance/ohibo/SVL1001C:messages:isevr:v1}EvrStatus, actual={urn:http://www.oracle.com/insurance/ohibo/SVL1001C:messages:isevr:v1}EvrStatuss**	Unknown or invalid (response) message: Wrong name of element

5 Appendix A – Provider web services documentation per service

Please use the WSDL that can be retrieved when a web service is deployed.

When services are changed the functional specification contains the latest changes.

Currently the provider web services offer no consistent read or locking functionality and can only retrieve data through a number of operations, except for services that implement changes.

The SVL_WS% packages can be used to get an overview of the existing services and their operations (the packaged procedures).

Each operation is documented through either HTML in the Description field (be sure you use an HTML editor to open this field) or is documented in a stored File (also present as 'File' in the application system SVL) that is named in the Description field.

Attention: Starting with release 2011.03.2 the PreAuthorization provider web service WSDL has been extended with an additional 'Herkomst' field. For the corresponding consumer web service the official Vecozo WSDL has been extended with an optional field that is not supported by Vecozo.

When the provider web service call specifies a value for the 'Herkomst' field the consumer call will fill the non supported Vecozo field. This means the call will not be accepted by Vecozo.

This functionality is introduced for the situation where an internal 'intervening component' (typical a middleware solution like a service bus) is used as intermediate and there is a requirement to distinguish between sources for pre authorization requests. This can be helpful in using the pre authorization web service for more source than the Vecozo pre authorization web service.