# Introduction to integrating Schenker’s e-services

Version 2.0

## Revision history

***Latest change first!!***

| Date | Version | Revision |
|---|---|---|
| 2008-06-18 | 2.0 | Rewritten to suit the technique of today. |
| 2000-03-03 | 1.0 | Document created. |

## Table of contents

## 1. Introduction

Schenker has internal systems managing information which are of interest for our customers/partners. Today several of these systems are accessible via Internet, so called Online-services. These systems does not only offer information exchange, but also an interface allowing customers/partners to integrate their systems with Schenker's.

Every service has an API who describes how a request should be built to work. Standard for these are WDR, Web Development Rules, which is Schenker's standard for how the request should look like. This document is about these rules.

All e-services are accessible with HQF, Http Query Format, which is Schenker's way of formatting requests and responses in raw data. HQF is sometimes also called text format.

XML can also be used to communicate with Schenker's e-services. Some services can only respond in XML while others can communicate both ways. XML is also based on the WDR standard. You will find accurate information on each service in their respective API.

The documentation includes information about variables of statistics and how they should be sent to the service. This statistics is used by Schenker for example when updating the services. Which customers using which services is then easy accessible and we can contact those who are affected by changes in their systems.

## 2. Communication

Schenker offers their customers/partners to access these on-line-services direct from their systems and not only via a web browser. This access is taking place using HTTP via Internet. Every service has it's own URL that can be found in the corresponding API's.

A call to a service is handled the same way as if it would have been called from a web browser. But the call is sent in a variable dictating if the answer should be HTML, XML or plain data.

### *2.1 HTTP Methods*

The HTTP protocol offers two main methods to send data. These are "GET" and "POST". When you send a request you will use one of them. We recommend that you if possible use POST. POST can send larger messages than GET and will send its information hidden from the user. When GET is used the user can see which variables being used in the request which makes variables such as passwords accessible from the outside. All our services requiring authorisation must only use POST. This is also mentioned in the API:s.

## 3. Character set / URL encoding

The character set used is Latin-1 (ISO 8859-1) which corresponds to the two first code pages in Unicode. The message should be recoded to become a 7-bits code according to URL encoding (RFC 1738). Example: if you send the field 'e_mail' with the value 'info@Schenker.Schenker.se' and the field 'code' with the value 'k%9S2!' to a service, the string will have the following format: "e%5Fmail=info%40Schenker%2ESchenker%2Ese&code= k%259S2%21".

More information regarding the character set and URL encoding can be found in a separate document.

## 4. Services

One central point in Schenker's web development is to always separate information, functionality and presentation from each other. A Schenker service has a well-defined API (Application Programming Interface).

There are several rules describing how a Schenker service must communicate. These rules will be described in this chapter.

## *4.1 The structure of the service*

The services are called the same way from an application as if it would have been called from a web browser. The response from the service has the same structure as the call. The fields are constructed the same way as a web browser build a 'GET' –message. The name of the field followed by a 'equals sign' (=), followed by the value of the field. If there are several fields, an 'ampersand' (&) is used to separate the fields.

Example: if you call a service with the fields 'foo' and 'bar' with the values 'foot' and 'bart' the string will have the following format 'foo=foot&bar=bart' when sent to the service.

## *4.2 Containers*

When communicating with a Schenker service the fields will be grouped together. These groups are called containers. To name a field in a container you put the container name first, then a full stop and lastly the name of the field.

Example: if the field 'page_size' belongs to the container 'request', the full name to describe the field is 'request.page_size'.

Some fields will belong to containers that in turn belong to other containers. Example: if the field 'message' belongs to the container 'error', which in turn belongs to the container 'system', the full field name will be 'system.error.message'.

## *4.3 Vektorer*

The string format also supports arrays. To describe a level in an array an index number is used starting on zero.

Example: You want to send an array with three delivery dates. The name of the array could be 'shipment_list' and the name of the date field could be 'delivery_date_time'. The first item in the array would then be represented as 'shipment_list.0.delivery_date_time'.

To complete the above example, lets say you want to send three delivery dates 2007-02-08 05:06, 2007-03-20 22:38 and 2008-01-01 00:00 to a service (without GMT zone specification). The final but not URL encoded field names and values would thus be as follows:

shipment_list.0.delivery_date_time=200702080506

shipment_list.1.delivery_date_time=200703202238

shipment_list.2.delivery_date_time=200801010000

The URL encoded and complete string would be:

shipment%5Flist%2E0%2Edelivery%5Fdate%5Ftime=200702080506&shipment%5Flist%2E1%2Edelivery%5Fdate%5Ftime=200703202238&shipment%5Flist%2E2%2Edelivery%5Fdate%5Ftime=200801010000

The values are not expected to adhere to any special order, i.e. the container could have the following format.

shipment_list.1.delivery_date_time=200703202238

shipment_list.0.delivery_date_time=200702080506

shipment_list.2.delivery_date_time=200801010000

## *4.4 Types of fields*

All fields must be of a special type and follow certain rules.

| **Field type** | **Explanation** |
|---|---|
| String | 1 GB sized string. Character set Latin-1. |
| Integer | +/- Integer 9 digits. |
| Float | Float represented with a decimal point, not a comma. If there are no decimals no point should be submitted. Example: '5.3' and '4', not '5,3' or '4.0'. If the field would contain a currency value the correct number of decimals will be returned, "5.00", "30.20" and "200800.00". |
| Date | The date and time format used is based on ISO 8601 Basic. The format is also compatible with the EDI-fact standard. The date must have the following format 'YYYYMMDD'.<br>**Example of the only approved date format:**<br>20070624 |
| Time | The time format used is based on ISO 8601 Basic. The format is also compatible with the EDI-fact standard.<br>Time must be written in the following format 'HHMMSS', i.e. '235959' (24-hour mode, not 'am' or 'pm').<br>Since different services can exist in different time zones, it's possible to specify the GMT zone by adding '+' or '-' (one exception, see below) followed by a four digit code '+0100' means one hour, 0 minutes east of the Greenwich meridian. A service positioned on the Greenwich meridian, which is GMT +/-0, can be written as '+0000', '0000' or 'Z' (i.e. '235959Z').<br>If a service receives a time without GMT zone specification, the time will be interpreted as being in the same GMT zone as the service.<br>A service that demands a GMT zone to be specified will return an error if a time is received without a GMT specification.<br>**Exemple of approves formats:**<br>113302+0200<br>113302-0200<br>113302Z<br>113302 |
| Date and time (time stamps) | The date and time formats used are based on ISO 8601 Basic. The format is also compatible with the EDI-fact standard.<br>The date and time specification is simple. By adding the time format to the date format.<br>**Example of approved formats:**<br>20070624113302+0200<br>20070624113302-0200<br>20070624113302Z<br>20070624113302 |
| Boolean | Used to describe true / false. Here the value is numerical. |

| | |
|---|---|
| | 0 = false, 1 = true |

## 5. Service containers

You have now seen the format of the communication rules and the notation for a service's information. You are now going to be introduced to the rules applying for service containers, used when the services are communicating with each other. All you learnt so far applies. Service containers are a good way of standardizing the structure of information.

### *5.1 Interaction*

When a service is called and receives fields and values, the service is expected to return all the calling information, plus the fields with their corresponding values generated from the service.

IMPORTANT!

The container concept reserves the right to return more information than what the receiver expects to receive. This is not a fault and the receiver should ignore the added information. This is a very important addition enabling minor changes/additions to the interface to be made without updating the client or server every time.

### *5.2 Basis containers*

There are four basis containers. These are 'request', 'response', 'data_list' (an array of containers) and 'system'. Consequently no field can only belong to ex. 'reference'. The field must be under one of the four basis containers, for example: 'request.reference'. The reason for having the basis containers is that it makes it easier to separate one kind of information from another. Below are the rules applying to the separate containers described.

#### The "request" container

When a service is called it accepts arguments which are not database related in this container, i.e. the type activity or diverge selections. The container is 'read only'.

Example of argument: "select.country_code" and "page_size".

#### The "response" container

This container contains information about information in the 'data_list' -container sent back from the service. The response container is 'write only'. If information is sent to the service in this container it will be ignored.

#### The "data_list" array

Observe that this is not a container but an array with containers!

The 'data_list' array contains the masses of information. For example, here you will receive the statusinformation from the Track & Trace service. The 'data_list' array is 'read and write' since it is also used to send information to a service. Information needed to be returned having nothing in mutual with the information mass, is sent back in the 'response' container, ex. 'data_list_count'. The 'data_list' array will always contain the same amount of elements as information containers, i.e. no empty elements in the 'data_list' array will be returned from the service.

#### The "system" container

The 'system' container will contain information about the system and its status. Error handling will for example be found here. The 'system' container is 'read and write', it sends information to the service (variables of statistics) and returns from it (error).

# 6. Predefined fields

Apart from the rules for the four base containers where are some predefined fields, these are mentioned here.

## *6.1 Basis containers*

Basis containers structure.

| **Name** | **Type** | **In/Out** | **Explanation** |
|---|---|---|---|
| request | Container | In | Field reserved for the calling service. |
| response | Container | Out | Information from a service |
| data_list | Vector | In/Out | The information mass |
| system | Container | In/Out | System information, for example: error handling |

Standard fields in the basis containers.

| **Field name** | **Type** | **Explanation** | **Mandatory** |
|---|---|---|---|
| request.service.action | String | Can be one of 'select', 'insert', 'update' or 'delete'. This field tells the service what it should do. Example: A booking service uses 'insert' and a service searching for goods uses 'select'. | No |
| request.service.method | String | Used when there are several types of methods, which a service can handle. Example: Different types of searches. | Yes |
| request.service.type | String | Detailed information clarifying for the service what it should do if a 'method' is ambiguous. | Yes |
| request.page_size | Int | The size of the data page. Each service, which allows 'request.service.action 'with the value' select, can accept 'page_size'. Default is 0 (infinite). | Yes |
| request.absolute_page | Int | The chosen data page. Every service permitting 'request.service.action' with the value 'select' can accept 'absolute_page'. Default is 1 (first page). | Yes |
| request.select | Alla | Containers containing selection criteria, ex.selection based on date, country etc. | No |
| response.service.name | String | The name of the service. | No |
| response.service.version | String | The version of the service in the format "33.12" or "2.1". | No |
| response.data_list_count | Int | The same as the amount of containers in the 'data_list' array. | No |
| response.total_record_count | Int | Equivalent to the number of posts found during a specific search using a service. If all posts are returned by the service, this number will correspond to the 'response.data_list_count'. For more information, see 'request.page.size' and | Yes |

| | | | |
|---|---|---|---|
| | | 'request.absolute_page'. | |
| system.error.id | Int | Error code 0 means ok. | No |
| system.error.message | String | Error message in English. Only for logging. | Yes |