Oracle® Retail WebTrack Configuration Guide
Release 12.0
May 2006

ORACLE

Contents

Preface

Oracle Retail's WebTrack is a web-based, collaborative critical path management solution which brings the members of a client's supply chain together to track critical, time-sensitive business activities and events. This Configuration Guide should serve as a reference for anyone using WebTrack to configure these elements throughout the application to ensure the most efficient business model.

Audience

Anyone with an interest in developing a deeper understanding of the configuration capabilities surrounding Oracle Retail WebTrack will find valuable information in this guide.

Related Documents

If you wish to find further information, see the following applicable Oracle Retail documents:

- Oracle Retail Design Online Help
- Oracle Retail Design User Guide
- Oracle Retail Design Operations Guide
- Oracle Retail Design Release Notes
- Oracle Retail Design Configuration Guide
- Oracle WebTrack Release Notes
- Oracle WebTrack Online Help
- Oracle WebTrack User Guide
- Oracle Retail Retail Server Installation Guide
- Oracle Retail Retail Server Data Model

Customer Support

- https://metalink.oracle.com

When contacting Customer Support, please provide:

- Product version and program/module name.
- Functional and technical description of the problem (include business impact).
- Detailed step-by-step instructions to recreate.
- Exact error message received.
- Screen shots of each step you take.

1
Introduction

In today's business climate, clients wish to take advantage of strategic opportunities. That is, they may wish to expand the ratio of import to domestic products, or shift to a more profitable private-label branding strategy. To accomplish these objectives, Oracle Retail WebTrack's ability to track and manage the development and global sourcing of goods becomes paramount to competitive success. WebTrack is a critical path management tool that binds the client with its multitude of trading partners (such as third-party agents, manufacturing suppliers, raw materials vendors, and so on) to manage the complex process of bringing goods to market.

The system provides a solution to issues raised around communication methods, supplier reaction time, workload balancing, and data management. Enhanced visibility and improved communication with supply chain partners brings order and control to the event management process. Using WebTrack to work collaboratively with trading partners, the client gains a greater ownership of events, reduces lead times, improves communication, and reduces costs through the supply chain.

Functional and Technical Capabilities

The system offers the following functional and technical capabilities:

- The web-based collaborative architecture enables everyone in the supply chain to have secure access to the same information, improving visibility, providing one version of the truth, and allowing for the proactive management of projects.
- The system's flexible, template-based process allows the client to determine the key events, dependencies, owners, and lead times that are used to manage processes. The system thus provides for both a mechanism for business process re-engineering and a consistent process approach throughout the internal and external community.
- Sophisticated dependency handling among events provides flexibility and allows clients to determine the complexity they would like to reflect within the track management process.
- Nested track functionality provides the flexibility to track all events with varying priorities within the same tool. That is, a parent track might be used to track the high-level events related to a process, and the nested child track could be linked to the parent track and used to track the lower-level events that also require tracking and visibility.
- The system's mass change capability provides an efficient means of maintaining track details, improving the overall accuracy and timeliness of the data tracked within the solution.
- Automated and manual diary entries provide improved controls and accountability over project management data. The ability to automatically send the diary entry as an email improves timely communication.
- By centralizing key event data in one place, the system allows the client to perform the following:
  - Evaluate the performance of events, tracks, and trading partners.
  - Report event management progress accurately.
  - Make strategic, fact-based decisions.

- Because the system utilizes some features of the Java 2 Enterprise Edition (J2EE) architecture, clients have a choice in their selection of databases and application servers.
- Market-proven and industry-standard application programming interfaces (API) are utilized (that is, RMI, JDBC, and so on).
- Java applications such as Oracle Retail WebTrack have enhanced portability which means Oracle Retail's clients are not 'locked' into a single platform. Upgrades are easier to implement, and hardware is easier to change.

WebTrack Administration Console

The WebTrack Administration Console provides the following functionality and more:

Options

Oracle Retail WebTrack provides Event Options. One option allows the blanking of revised dates on an event if the revised date is the same as the planned date. A second option is the enabling of an event confirmation column in the Track Details screen.

Mail

The mail administration is used to configure automatic email alerts to users based on various events.

Lists

Extra fields support configuration of non-standard elements in the integration of Oracle Retail Design to Oracle Retail WebTrack.

Users

Oracle Retail WebTrack provides a local type in the WebTrack user permissions to enable a button to launch Oracle Retail Design from the Track List or Track Details screens.

WebTrack User Console

The WebTrack User Console provides the following functionality and more:

Diary Entry

A 'Track Name Changed' diary entry has been introduced for changes made to a track name through the integration configuration between Oracle Retail Design and Oracle Retail WebTrack.

Track Details

When enabled by the administrator, a column is available in the Track Details screen to show the confirmation of a plan or revised date by an event owner.

Track List Screen

When enabled by the administrator and mapped through the Oracle Retail Design configuration file, the user is able to add the season attribute and the design vendor (Extra field) to the Track List screen. In addition, Oracle Retail WebTrack allows the user to launch Oracle Retail Design from the Track List screen.

Mail

When added to a mail message, the user is able to launch the Oracle Retail Design application from the mail message via the Oracle Retail Design URL

A Template-Based Approach

The client defines a template to include specific events and associated lead times. Events are specific tasks that the client monitors for completion. Events that are commonly used together may be grouped into a template. Using the template and supporting foundation data such as project or purchase order (PO) data, the client creates a ‘track’. A track provides a mechanism for communicating expectations, schedules, and event assignments to all members of the supply chain including trading partners. In sum, to help save time within supply chain processes, Oracle Retail WebTrack uses tracks to manage the events associated with projects and, on a more complex level, purchase orders. Because of the flexibility associated with Oracle Retail WebTrack, a client could use the application to support the pre-production tracking of a new product, the opening of a new store, or the production and logistics tracking of items on a purchase order.

Individual event owners are accountable for managing specific events within the track. Access to all tracks in one place improves visibility to changes. Automatic email alerts provide reminders to event owners if events become late or overdue. As updates and changes are made to the track, a diary of all activity is logged. Track details and summaries can be printed from the tracks and reports windows in WebTrack, and the system’s robust reporting abilities on track data allow the client to report on progress and to manage by exception.

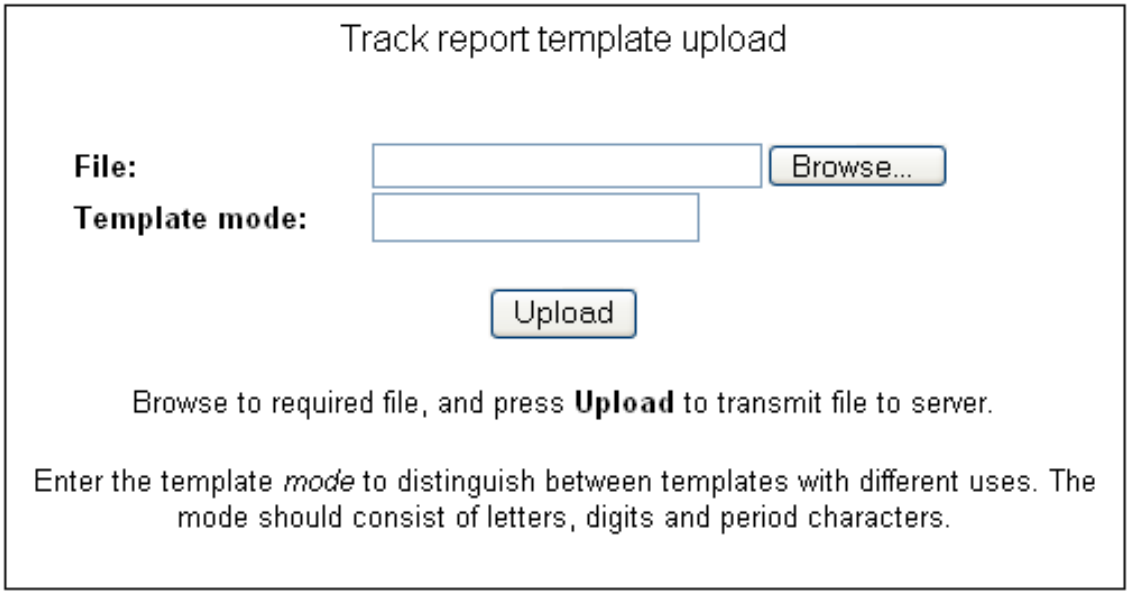
Server Side Reporting Template Administration

Oracle Retail WebTrack supports the generation of printable output on the client and server side. The client side printing is generated based on specified rules within the code. If users are expecting an exact format, they may not choose to leverage the client side print. Format files can be developed using XFO technology and can be uploaded by browsing to the following URL from within the Oracle Retail WebTrack Administration Console:

- https://www.retail.com/applications/design/template.jsp

The administrator is prompted to browse and upload the template file, define the format that it uses, and identify a mode that is used to cross-reference the configuration file. The template mode is a free-form text field and the value input is used to cross-reference the uploaded file in other configuration administration steps. Specifically, the mode is used within the general user view configuration file to identify a format that can be used by the server side printing options available to the user. In addition, the mode can be referenced within Oracle Retail Integrator as the ObjectiveSheetType during the setup of the run type used to support the technical specification export process.

Note: Although the objective sheet template upload process continues to support the upload of .xmf files, Oracle Retail recommends that xfo formats be used.



Track report template Upload window in Oracle Retail WebTrack

The actual development of the XFO files can be done in any text editor. A text editor that supports XML would be most efficient and is recommended. The primary functions supported within the development of the XFO files are highlighted in “Chapter 2 – XFO templates”.

2
XFO Templates

The template driven PDF generator used to format print files on the Oracle Retail WebTrack client is named “XFO.” An XFO template is an XML file containing a mixture of markup XML and Style File XFO processing elements which are used to control the output and to include dynamic values.

The first implementation uses the XSL formatting objects (XSL-FO) markup language, in conjunction with Apache FOP (http://xml.apache.org/fop), which is a XSL-FO to PDF renderer. FOP implements most of the XSL-FO standard, but there are some limitations – see documents on the above website for conformance details.

XFO Operation

The XFO processor first reads and parses the template file. Any XML errors found at this stage are reported by an error PDF produced by the processor. When a PDF is generated from a set of styles, the ‘style file’ elements are processed to produce pure XSL-FO output, which is passed directly to the FOP engine for rendering to PDF.

Basic Structure

An XFO template contains elements from the XSL-FO namespace and the ‘style file xfo’ control namespace. Conventionally prefixes **fo:** and **sf:** are used for these namespaces.

```
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format"
         xmlns:sf="http://www.retail.com/XSL/style-files">
```

The namespace URIs should be included exactly as above. If you need to emded SVG for advanced graphics, add the SVG namespace to the root element or the svg element when used:

```
<svg:svg xmlns:svg="http://www.w3.org/2000/svg”>
```

Expressions and Attributes

Attributes in **fo:** (and **svg:)** elements may contain expressions enclosed in ${ and }. These expressions are evaluated during the processing phase and final attribute value is passed to the XSL-FO output. Expressions are written using the standard ‘spec sheet’ expression language (see *expressions.doc*) and may refer to variables set earlier in the processor.

Example:

```
<fo:simple-page-master master-name="one"
     page-height="${pageheight}${unit}"
     page-width="${pagewidth}${unit}">

<fo:block break-before="${index > 0 ? 'page' : 'auto'}">
```

In the above example, the page dimensions in the page master as set using previously defined numbers and a unit string (mm, cm, in, etc). In the block element, the value of the break-before attribute is set to page or auto according to the value of the index variable.

Note that in common with all XML files, any & or < characters in expressions must be written by the character entities & and <

SF Processing Elements

The following **sf:** processing elements are available.

Attributes which are *expressions* are evaluated directly – they are *not* enclosed in ${ }. An attribute defined as a *string* may contain ${ } expressions.

An attribute defined as a *name* represents a 'variable' name. It must follow the rules for Java identifiers – essentially, the first character must be a letter or _ and the remainder can be letters, digits or _.

Any **sf:** element which contains other elements defines a new 'context'; variables defined in this context are not in scope outside it.

sf:str

`<sf:str x="expression"/>`

Evaluate the expression and include the result as a string at the current point in the XSL-FO output.

Example:

```
<fo:block><sf:str x="item -> shortname"/></fo:block>
```

sf:int

`<sf:int x="expression"/>`

Evaluate the expression and include the result as an integer at the current point in the XSL-FO output.

Example:

```
<fo:block><sf:int x="item -> quantity"/></fo:block>
```

sf:float

`<sf:float x="expression" fmt="string"/>`

Evaluate the expression and include the result as a decimal number at the current point in the XSL-FO output. The optional `fmt` attribute can be used to supply a format for the conversion (see **java.text.DecimalFormat**); if omitted a default format with 2 decimal places is used.

Example:

```
<fo:block><sf:float x="item -> elctarget" fmt="0.000"/></fo:block>
```

sf:date

```
<sf:date x="expression" fmt="string" tz="string"/>
```

Evaluate the expression as in internal date value (a Java time stamp divided by 1000), convert the date to a string using the supplied format or a default, and include the result in the XSL-FO output. See **java.text.SimpleDateFormat** for details of the optional format string. If the format is omitted a locale-specific date format is used.

The optional `tz` attribute is used to select the time zone for the formatting. It must be one of the time zone IDs understood by **java.util.TimeZone** (that is "Europe/London" or "PST"). If `tz` is omitted, the server's 'standard' time zone is used. Most dates in design (initial availability, bid deadline, etc) represent a 'day' and are stored internally using the standard time zone; `tz` should not be used with these. Dates which represent a point in time (last change, log dates, etc) contain a time of day element and the time zone is relevant.

Examples:

```
<sf:date x="item -> biddeadline"/>
<sf:date x="item -> changedate" tz="${tz}" fmt="yyyy-MM-dd HH:mm:ss"/>
```

The first example displays a date using the default format; the second displays a time stamp using such as 2004-11-09 12:23:13 with a timezone obtained from a variable.

sf:set

```
<sf:set n="name" x="expression"/>
```

or

```
<sf:set n="name" x="expression">
  ... content ...
</sf:set>
```

The expression is evaluated and assigned to the name. In the first syntax, the name remains in scope until the end of the current context. In the second example, the name is in scope during the processing of the embedded content.

Example:

```
<sf:set n="unit" x="'mm'"/>
<sf:set n="temp" x="x * 10>
   ... content ...
</sf:set>
```

The first example sets `unit` to the string `"mm"`; the second defines `temp` for the processing of the enclosed content.

sf:update

```
<sf:update n="name" x="expression"/>
```

or

```
<sf:update n="name" x="expression" index="expression"/>
```

The expression is evaluated and assigned to the most recent definition of the name. The name should have been defined by an earlier **sf_set** element. In the second form, the index expression is evaluated and used to set an elemente in the array identified by the name. The array should have been created earlier using the **array** function.

Examples:

```
<sf:update n="count" x="count+1"/>
```

Set `count` to its previous value, plus 1.

```
<sf:set n="arr" x="array(10)"/>
…
<sf:update n="arr" x="1" index="i+1"/>
```

Set element i+1 in the array arr to 1.

sf:func

```
<sf:func n="name" x="expression"/>
```

or

```
<sf:func n="name" x="expression">
  ... content ...
</sf:set>
```

Define the expression as a function. The rules for the scope of the name are as in **sf:set**.

Example:

```
<sf:func n="imwidth" x="$1 * min(1, min(maxwidth/$1, maxheight/$2))"/>
```

sf:if

```
<sf:if x="expression">
   ... content ...
</sf:if>
```

If the expression is defined and non-zero then process the content; otherwise ignore the content.

Alternative format:

```
<sf:if x="expression">
  <sf:then>
    ... content 1 ...
  </sf:then>
  <sf:else>
    ... content 2 ...
  </sf:else>
</sf:if>
```

In this form, the content in the ‘then’ part is processed if the expression is non-zero; otherwise the content in the ‘else’ part is processed. The ‘else’ part can be omitted but that is the same as the more succinct first format.

sf:for

```
<sf:for n=″name″ to=″expression″ from=″expression″ by=″expression″
     while=″expression″ set=″expression″ var=″name″>
  ... content ...
</sf:for>
```

sf:for is used to process content repeatedly. All the attributes are optional, but at least one of two, `while` or `set` must be used. To prevent the server running for ever as a result of a faulty template, a limit of 8192 iterations is imposed by the processor.

There are three distinct forms of iteration; any combination may be used:

1. Numeric
   a. Iterate over the range ‘from’ to ‘to’ inclusive, in steps of ‘by’. If ‘from’ is omitted, the iteration starts at 1; and ‘by’ is omitted the step is 1. If the name ‘n’ is supplied, the iteration value is assigned to it during the loop. This is roughly equivalent to the java loop, except that if the step ‘by’ is negative, the loop counts down and the test is named >= to. That is:
      `for (name = from; name <= to; name += by)`
2. Conditional
   a. If while is used, the loop terminates as soon as the expression evaluates to ‘false’ (undefined or zero).
3. Set
   a. The set expression should evaluate to a set of items; the loop continues whilst there are elements in the set; the current item is assigned to the name defined by var, if present. The item set will be defined outside the processor.

Examples:

```
<sf:for to=″10″>
  .. content ..
</sf:for>
```

Process the content 10 times.

```
<sf:for n="index" from="0" set="items" var="item">
  ... content ...
</sf:for>
```

Process the content over the set of items; the current item is assigned to `item`; the variable `index` counts up from zero.

sf:macro

```
<sf:macro n=″name″>
   ... content ...
</sf:macro>
```

Store the content against the name for later use. The macro is expanded using the **sf:call** element. Macros are useful for repeated header components, etc.

sf:call

```
<sf:call n=″name″>
   <sf:set n=″name1″ x=″expression1″/>
   <sf:set n=″name2″ x=″expression2″/>
   ...
</sf:call>
```

Process the content of the macro *name*; while processing `name1`, `name2` ... are set to `expression1`, `expression2` ....

The nested **sf:set** elements are optional.

Builtin Values and Functions

The processor always defines the variable now as the current date. This can be used to include the time of printing in a footer, that is:

```
<fo:block font-size="5pt">
  <sf:date x="now" fmt="yyyy-MM-dd HH:mm:ss" tz="Europe/London"/>
</fo:block>
```

The following functions are always available:

array(n)

Create a 1-dimensional array of size n. The size must not be more than 512.

Example:

```
<sf:set n=″arr″ x=″array(size+1)″/>
```

geticon(string)

Lookup the ‘icon’ named by the argument. The result is *undefined* if the icon does not exist, and an icon object otherwise:

Field	Type	Meaning
file	string	Image file name
width	integer	Image width
height	integer	Image height

Example:

```
<fo:external-graphic src="url(${geticon('logo') -> file})"/>
```

getprop(prop) or getprop(prop, deflt)

Lookup a *property* passed to the processor. If the property is not defined, the result is *undefined* or **deflt** if there are two arguments.

Example:

```
<sf:set n="pagewidth" x="getprop('pagewidth', 210)"/>
```

valuekey(v)

A value which is derived from a parameter lookup (that is the value of a mapped dropdown list box) may have an associated *external value*. This function returns the external value of v, if any, or v if there is no external value.

hasmorevalues(set)

The argument must evaluate to a 'set' (see sf:for); return 1 if there are further items in the set or zero if the set is exhausted. This can be used to determine whether a break is needed after an item, that is.

XFO and Tracks

When processing WebTrack tracks, there are additional predefined values and functions.

Values

trackcount

This is the number of tracks being processed and should not be used to iterate over the set – use tracks (below). The count may be used to estimate layouts, and so on,, but is not guaranteed to be accurate. It is possible that one or more of the tracks selected by the user have been deleted by another user between selection and processing.

tracks

`tracks` is a set of style objects. Iterate over the set with an **sf:for** element (see above).

Each object contains the fields shown below. Boolean fields are represented as a *combined* value with a numeric par of 0 or 1 and a string part of "false" or "true".

Field	Type	Meaning
name	string	Track name
ownerid	integer	Internal ID of track owner
ownername	string	Name of track owner
owneremail	string	E-majl address of track owner
toplevelenterpriseid	integer	ID of enterprise owning top level parent track
ownerenterpriseid	integer	ID of enterprise owning track
ownerenterprisename	string	Name of enterprise owning track
departmentid	integer	Internal ID of track department
departmentname	string	Track department
departmentnumber	string	Track department number
divisionid	integer	Internal ID of track division
divisionname	string	Track division
divisionnumber	string	Track division number
state	integer	Track state[1]
alert	integer	Track alert state[2]
enddate	date	End date of track
creationdate	date	Creation date of track
modificationdate	date	Modification date

Field	Type	Meaning
suspenddate	date	Date of track suspension; unset if track is not suspended
seasonid	integer	Internal ID of track season; zero if season is not set
ordernumber	string	PO or project number
ordered	integer	Internal ID of order
orderitemid	integer	Internal ID of order item
quantity	integer	Track quantity
value	decimal	Track value (quantity * item value)
orderinfo	string	Order information string
supplierid	integer	ID of supplier; zero for projects
suppliername	string	Name of supplier enterprise
accountid	string	Contact ID in supplier
itemid	integer	Internal ID of associated 'item' (or project)
itemtype	integer	Item type (1 = item, 2 = project)
itemname	string	Item or project name
stylenumber	string	Item style number
vendornumber	string	Item vendor number
colourid	integer	Internal ID of track colour, or zero
colourname	string	Colour name, or unset
firstopenid	integer	Internal ID of first open event
firstopenname	string	Name of first open event
firstopeneventid	integer	Event type ID in first open event
firstopendate	date	Date in first open event
firstopenenterprise	integer	Enterprise ID in first open event
firstopencontact	integer	Contact (user) ID in first open event
orderattributges	attribute array	Array of the attributes associated with PO/project
events	event array	Array of event objects
diary	diary array	Array of diary entry objects

Note: The *state* field is 0 for active tracks, 1 for archived tracks and 2 for cancelled tracks. The *alert* field is 0 for green, 1 for orange and 2 for red. ID values (such as *seasonid)* can be used for parameter lookups. They are internal values and have no external meaning.

Subsidiary Objects

Attribute

The value of an **attribute** object is the attribute value, as a number or string (depending on the attribute definition). The object also contains these fields:

Field	Type	Meaning
name	string	Attribute name
key	integer	Internal attribute key
format	string	Attribute display format

The *key* is the internal key for the attribute. For attributes set by the configurable Design to WebTrack project creation interface, the key is (60000+2+N) where N is the integer in the `attr.N.name` property. The standard attributes with keys 60001 and 60002 are always set to the internal style ID and supplier name.

The format string is **t** (or unset) for text values, **i** for integral values, **fN** for decimal values (displayed with N decimal places, 2 by default) or **d** for date values.

Event

An event object represents an event in a track. It contains the following fields:

Field	Type	Meaning
pathid	integer	Internal ID of track
pathowner	integer	Internal ID of track owner
pathenterpriseid	integer	ID of track enterprise
eventid	integer	Internal ID of event
sequencing	integer	Event sequencing option[1]
done	boolean	True if event is complete or cancelled
cancelled	boolean	True if event was cancelled
plan	date	Plan date
revised	date	Revised date
actual	date	Actual date
amberdays	integer	Orange alert threshold
reddays	integer	Red alert threshold
reminderdays	integer	Event reminder threshold
alert	integer	Event alert state (as for tracks)
eventnameid	integer	Internal event type
ownerenterpriseid	integer	ID of event enterprise
ownerenterprise	string	Name of event enterprise

Field	Type	Meaning
owneruserid	integer	ID of event user (contact)
ownername	string	Name of event user
owneremail	string	E-mail address of event user
flags	integer	event flags[2]
nestedpathid	integer	ID of nested track, or zero

Note: The *sequencing* field is zero if the event has no dependencies, '1' if the dependency is 'after previous', '2' if the dependency is 'after all previous,' or negative for more complex dependencies. The bottom section of the *flags* field is set if the event is 'confirmed'.

Diary

A **diary** object represents an entry in a track diary. It contains the following fields:

Field	Type	Meaning
pathid	integer	Internal ID of track
action	integer	Action type of entry
when	date	Date entry was made
text	string	Entry text (may be empty).
eventid	integer	Internal ID of associated event
eventname	string	Name of associated event
eventnameid	integer	Internal type of associated event
eventdate	date	New date for event (may be unset)
userid	integer	Internal ID of user making entry
username	strintg	Name of user
useremail	string	E-mail address of user
userenterpriseid	integer	ID of user enterprise
userenterprise	string	Name of user enterprise
emails	string array	Extra emails in entry

The *action* field identifies the update associated with the diary entry. It can be used to filter out unwanted entry types.

Action	Meaning	Action	Meaning
0	Manual user entry	14	Track reinstated
1	Event completed	15	Event deleted
2	Revised date changed	16	Track quantity changed
3	Revised date accepted	17	Track created in suspended state
4	Revised date rejected	18	Track suspended
5	Track created	19	Track unsuspended
6	Track archived	20	Plan date changed
7	Event cancelled	21	Plan and revised dates changed
8	Event added	22	New template applied
9	Event contact updated	23	Event auto-cancelled
10	Track owner updated	24	Track name changed
11	Track split (old form)	25	Event confirmed
12	Track split (new form)	26	Event unconfirmed
13	Track cancelled		

To generate suitable text for display, use the format diary function (see below).

Functions

There are additional functions available to obtain more complex track values. As well as the functions listed below, the standard parameter lookup and formatting functions are also available. The parameter lookup functions work with the parameters defined in the enterprise of the current user. This is generally most useful for ‘my tracks’ displays.

getitemextra(track, number)

Get one of the numbered ‘extra’ values from the item (product or project) associated with the track.

Example:

getitemextra(track, 1)

getpartnerinfo(track)

Get a 'partnerinfo' object for the supplier associated with the PO track. The object contains the following fields:

Field	Type	Meaning
accountnumber	string	Account number for the supplier enterprise
code	string	Partner code string
info	string	Partner info string

The result is *undefined* if the track was created from a project.

findattribute(track, keyorname)

Search for an order attribute on the track. This is a convenience function for accessing the **orderattributes** field in a track object. If the second parameter is a number, the search is against the attribute 'key'; otherwise the search is against the attribute name. The result is an **attribute** object, or 'undefined' if the attribute is not found.

Example:

```
indattribute(track, 60001)
```

Find the Design style ID attribute for a track.

formatdiary(track, diaryentry)

Generate descriptive text from a diary entry object. The text is produced from the same resource strings as are used to display the diary in the client code.

Configuring XFO Printing in WebTrack

XFO printing is now available in the WebTrack client as an alternative to the rather basic client-side PDF generation. XFO is the only solution for track diary printing. Templates are uploaded by the administrator from:

```
http://server/applications/tracks/template.jsp
```

Templates may be tested using the standalone 'xfotracks' application.

To configure XFO printing for tracks, select the **Print Templates** tab in the WebTrack administration **Options** window. Enter a set of print modes, one per line. The syntax is similar to the *modes* parameter in print setup for Design.

That is:

```
Track Summary/tracksumm,prop.type=summ,sel=all
Track Detail/tracksumm,prop.type=detail,sel=either
Track Diary (text mode)/diary,prop.format=text,max=1000,prop.diary=1
Track Diary (list mode)/diary,prop.format=list,max=1000,prop.diary=1
```

The string before the / on each line is shown in the drop down in the print dialogue box. The string after the / is the mode string used to find the file name.

The mode strings may be followed by options and properties which are passed to the formatting engine. In the example here, the same template (*xfo-diary-19* or *xfo-diary-0*) is used for two modes; tests within the template will control whether the output is in text or list format. Properties (starting with *prop.*) can be retrieved in the template using the **getprop** function.

The property **prop.mine** is set automatically to 1 for ‘my tracks’ displays or 0 for ‘other tracks’. It can be used in the template to show or hide information depending on whether the user owns the tracks.

There is an additional overhead when loading diary entries for printing. If the template requires access to the diary, include the **prop.diary** property with value 1. In the example above, the first two modes do not use the diary; the second two do.

There are also some built in options which are interpreted in the client.

Option	Value	Default	Meaning
max	Integer		Maximum number of tracks supported by the mode; can be used to prevent over-complex reports being generated for a large number of tracks.
sel	*list*, *all* or *either*	*list*	If *list*, the tracks which are selected in the list are printed; if *all*, all the tracks in the (filtered) list are printed, irrespective of the selection; if *either*, the selected tracks are printed, or the entire list if there is no selection.

3

Spreadsheet Expression Syntax

This chapter describes spreadsheet expression syntax used throughout the configurable definition files of Oracle Retail WebTrack. Expressions can be leveraged in user view configuration files to support the Oracle Retail Design to Oracle Retail WebTrack project integration, in the tab layout definition files to support configuration sheets, and in XFO templates used to support printing and export processes.

Data Types

Values in expressions are either numbers or strings.[1] A value can also be undefined (represented internally by the special numeric value NaN, or not-a-number). When an undefined value is used in an expression, the result is generally also undefined. There are built-in functions which will test for undefined values.

Numbers are stored in double precision format, with an approximate range of ±5-324 to ±2+308. A numeric constant is a sequence of decimal digits, with an optional decimal point and exponent. The exponent part is e±integer (or E±integer). If the sign is omitted, a positive exponent is used.

Examples:

```
1.  1.0  .023   1e4 1E-10   2.3e+5
2.  3e+5 = 2.3x105 = 230000.
```

A string constant is a sequence of characters enclosed in ‘ or “ quotes. The quote character used to start a string must be used to end it (when entering a string constant in an expression used as an XML attribute, avoid using the quote character used for the attribute).

Within a string the backslash (\) character is used to introduce escape sequences. The following sequences are useful:

Sequence	Character
\\	\
\”	“
\’	‘

Examples:

```
“abcd” ‘Please type something’       “A string with a quote \” inside”
```

[1] Values derived from user entry fields can sometimes be *both* a number and a string. That is a country dropdown in Design will be linked to a value that contains the internal ID of the country as a number and the display name of the country as a string. The numeric value is available for parameter lookups, whilst the string value is used for display in text fields, and so on.

Lists

Value list objects are returned by the **lookup** functions. They are generally used to populate drop down choices in spec sheets and to populate dynamic row and column sets in spec sheet matrices.

The number of items in a list can be determined with the **length** function and individual elements can be obtained by subscription.

Arrays

An array value has any number of dimensions. The number of elements is $b_1 \times b_2 \ldots$ where b_i are the bounds of each dimension. The bounds may be obtained using the **length** function and individual elements may be obtained using subscription.

Arrays are used to represent cells in the dynamic row/column areas of matrices in spec sheets. A cell which is either in a dynamic row or column area (but not in both) is represented by a 1-dimensional array, while a cell which is both areas is represented by a 2-dimensional array.

Arrays are also used to represent spec sheet mappings attached to entire forms or matrices, or matrix rows or columns.

When such an array cell is used in an expression, the context of the expression is taken into account to determine which element(s) of the array are involved. If the cell is used in a dynamic array context with the same "dimensions"[2], only a single element will be selected and a normal scalar expression is evaluated.

That is, if *x$3.4* represents a cell in the dynamic area of a matrix, and the expression:

```
'x$3.4 * 2'
```

is used in a dynamic area in another matrix with matching dimensions, the expression is evaluated once for each element in the array and the result used for the matching element in the destination matrix.

If an array cell is used in a context which has no dimensionality (that is, is not part of a dynamic area in a matrix) or which has different "dimensions", then the array value as a whole is used. In this case, the only valid use of the cell is for aggregation or subscription.

That is, using the array cell *x$3.4*, the expression:

```
'sum(x$3.4)'
```

could be used in a numeric field in the spec sheet to display the sum of all the elements in the cell.

[2] In other words, the "dimension" attribute strings in the dynamic area definitions in the two matrices are equal.

Array Items

A single-dimensional array may be used directly in an expression by enclosing a list of values in { } brackets. That is:

```
{ 1, 2, 3, 4, x+y, 'string'}[2]
```

has the value 2. Array items are useful in conjunction with loops in XFO templates.

Object Values

An object value is analogous to a Java object with public fields. The value is constructed by client code (often automatically from a real Java object) and made available to the expression evaluation context.

The fields in an object value are accessed using the -> operator. The right hand side of this operator must be a name.

That is, assuming that a **java.awt.Point** object has been mapped into an object value and stored in 'pt':

```
pt -> x
pt -> y
```

will extract the two fields.

Variable Names

Variable names contain letters, digits, underscore (_), dollar ($) or period (.) characters. A name should start with a letter or digit (names starting with other characters are reserved for internal use).

The special name *$n*, where *n* is a decimal number, represents a function argument. It has no meaning outside of a function definition. $1 is the first argument; $2 is the second, and so on. $0 represents the number of arguments in the call.

Function Calls

A function call is a reference to a built-in or user-defined function. A user defined function can be used to replace commonly-used expressions by a simple call.

The syntax of a call is:

`functionname(arg1, arg2, …)`

There may be zero or more arguments.

Aggregate Functions

Some of the built-in functions operate by *aggregating* the arguments. These functions treat arrays, lists and iterators specially by including all their elements in the aggregation. That is, if a and b both represent arrays, then `sum(a, b)` will sum all the elements in both arrays.

Expressions

Names, constants, and function calls are combined into expressions using *operators*. Operators have differing *precedence.* Higher precedence operators are evaluated before lower precedence operators. Parentheses (()) may be used to alter the order of evaluation.

Operator	Precedence	Meaning
?:	1	Conditional expression
|	2	OR – the result is 1 if either operand is non-zero, and 0 otherwise
&	3	AND – the result is 1 if both operands are non-zeo and 0 otherwise
= or ==	4	Equals: evaluates to 1 or 0
!= or <>	4	Not equals
<	5	Less than
<=	5	Less than or equals
>	5	Greater than
>=	5	Greater than or equals
+	6	Addition or concatenation
-	6	Subtraction
||	6	String concatenation
*	7	Multiplication
/	7	Division
%	7	Modulus (remainder)
^	8	Power: a^b = a^b
->	9	Field selection

The addition and comparison operators may be used with string operands; if one operand is a number and the other is a string, the number is converted to a string before evaluation. The addition operator (+) performs string concatenation if either operand is a string; the concatenation operator (||) always converts *both* arguments to strings before evaluation.

That is, 1+2 evaluates to 3, while 1 || 2 evaluates to “1.02.0”.

Other operators evaluate to *undefined* if either argument is a string.

Conditional Expressions

The conditional expression operator ? is used with three operands:

```
a ? b : c
```

If a is non-zero the result is b otherwise the result is c.

Subscription

Individual elements of arrays and lists may be obtained using subscripts in [] brackets. Arrays with more than one dimension require multiple subscripts separated by commas. All subscripts are zero-based. An alternative is to use the **element** function.

Iterator Expression

Iterator expressions can be used to perform an aggregate calculation with an expression calculated over all the elements of an array. Iterators are recognized only as the arguments to aggregate functions such as **sum** or **avg**.

The selection of elements in the array is performed by the evaluation client, possibly using constraints to limit the set returned.

The syntax is:

```
{ name : expression }
```

The expression is evaluated over all the elements of the array represented by ‘name’. The value is *undefined* if the client does not support iteration over the array.

Iterators are commonly used in spec sheet matrices to perform some complex aggregation over the elements in a dynamic row or column set.

That is, assuming that *x$3.4* represents a matrix cell in a dynamic region:

```
sum({ x$3.4 : x$3.4 ^ 2 }
```

will sum the squares of all the values in the set.

```
avg({ x$3.4 : lookup(‘parameter’, x$3.4))
```

will average the results of the lookup call over all the elements in the set.

Examples

Expression	Notes
`1+2`	
`1+2*3^4`	This is equivalent to 1 + (2 * (3^4))
`((1+2)*3)^4`	
`a ^ 0.5`	The square root of a
`a = b`	If a equals b then evaluate to 1 otherwise evaluate to 0.
`a = b & c = d`	If a equals b and c equals d then evaluate to 1 otherwise evaluate to 0.
`sum(a,b) < 10 | c >= 5`	If sum(a,b) is less than 10 or c is greater than or equal to 5, then evaluate to 1 otherwise 0.
`val3 = 10 ? a+b : a-abs(zz)`	If val3 equals 10, then the result is a+b; otherwise the result is a-abs(zz)
`x[i]`	The i'th element in the array or list x.
`y[1, e+7, n]`	An element in the 3-dimensional array y.
`value -> name`	The field 'name' in the object value represented by 'value'
`value[i] -> name` `(value[i]) -> name`	The field 'name' in the object value stored in the i'th element of the array 'value'.
`value -> items[x]` `(value -> items)[x]`	The x'th element in the array stored in the field 'items' in the object value 'value'.
`avg({ x$3.4 : abs(x$3.4) }`	Average the absolute value of the elements in the array represented by x$3.4.

Built-In Functions

A number of built-in functions are available for use in expressions. Some are available in all contexts; some are specific to spec sheets in all contexts[3], and some are specific to Oracle Retail Design. If a function is used with an incorrect number of arguments, or arguments of the wrong type, the result is *undefined*.

Functions Available In All Contexts

Function	Arguments	Meaning
number(a)	Number	Return a as a number. This is used for 'combined' values which would otherwise be used as strings in expressions.
floor(a)	Number	Return the largest integer which is not greater than a: floor(1.9) = 1 and floor(-1.9) = -2
ceil(a)	Number	Return the smallest integer which is not less than a: ceil(1.1) = 2 and ceil(-1.1) = 1
round(a)	Number	Round a to nearest integer
abs(a)	Number	Absolute value of a
isset(a)	Any	If a is defined, return 1; otherwise return 0
ifset(a, b)	Any	If a is defined, return a otherwise return b
zerop(a, b)	Numbers	If a is zero, return 0 otherwise return a*b; this function is useful because b need not be defined if a is zero.
if(a1, b1, a2, b2 ..)	Any	If a1 is non-zero, the result is b1; otherwise if a2 is non-zero, the result is b2, and so on. If none of the conditions succeed, the result is undefined if there is an even number of arguments, or the last argument if there an odd number of arguments. if(a, b, c) is equivalent to (a ? b : c).
length(a) § or length(arr, index)	Array, list or string.	In the single argument form, return the length of the one dimensional array, list or string a. If a is a string the length is the number of characters in the string. In the two argument form, return the length of the dimension index in the array arr. If arr is not an array the result is undefined.

[3] That is, within the Spectrum spec sheet application.

Function	Arguments	Meaning
substr(str, m) or substr(str, m, n)	String and numbers	This is equivalent to the Oracle SUBSTR function. The first argument is converted to a string. The result is the substring starting at position m which is n characters long. If n is omitted, the remainder of the string is returned.
indexof(a, b) or indexof(a, b, c)	Strings	In the two argument form, return the position of the substring b in the string a, or -1 if the substring is not found. In the three-argument form, start the search at position c. This is analogous to a.indexOf(b) or a.indexOf(b, c) in Java.
lower(a)	String	Convert the string a to lower case.
upper(a)	String	Convert the string a to upper case.
element(a, x_1, ..)	Array or list	Return the element with subscipts x_1, … from the array or list a. This is exactly equivalent to a[x_1, …] - the [] subscripting syntax was introduced after element.

Aggregate Functions

Function	Arguments	Meaning
sum(a, b, c …)	Numbers	Sum all the arguments; if any are undefined the result is undefined.
zsum(a, b, c …)	Numbers	Sum all the arguments, ignoring any that are undefined.
avg(a, b, c …)	Numbers	Average of the arguments; if any are undefined the result is undefined.
zavg(a, b, c …)	Numbers	Average of the arguments, ignoring any that are undefined.
max(a, b, c …)	Numbers or strings	Maximum value of all the arguments; if any are strings, the result is a string otherwise the result is a number.
min(a, b, c …)	Numbers or strings	Minimum value of all the arguments.

4
Event Import

Elements and Attributes

This section provides a reference for elements and attributes used in defining WebTrack Event Import layouts. Elements are presented in the order in which they appear in the eventimport.dtd file. For each element, a description, format, and example (where applicable) are provided.

Identifying Statement

Each event import sheet starts with a statement in the following format:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Event

The event element contains the information required to perform updates on a Track's events. The subordinate elements are used to identify an order event versus a project event, as well as the action to take on the event.

Events is the enclosing element. The enterprise ID identifies the enterprise doing the importing (that is, a Colby Enterprise), NOT necessarily the enterprise owning a style. It is expected that integrator adds this attribute automatically based on the source of the file.

The version attribute is optional.

```
<!ELEMENT events (event+)>
<!ATTLIST events
   version CDATA #IMPLIED
   enterprise CDATA #IMPLIED
```

Multiple event elements can be included within the events element.

Multiple tracks can be updated within the same XML file as well. The action field determines the type of action the sending enterprise is trying to perform.

Valid values are "A"dd, "U"pdate and "D"elete. You currently cannot add or delete an event.

The track type is the type of track to be updated. The two valid values for track types are "P"roject and "O"rder.

The track owner represents whether the sending enterprise is the track creator or not.

> **Note:** If enterprise A creates a nested track, then some child tracks of this track could be owned by enterprise B. But enterprise A is the creator of these child tracks.

```
 -->
<!ELEMENT event (partner_code?, order?, project?, parent_name?,
child_name?, dates?, diary_comments?)>
<!ATTLIST event
   action NMTOKEN #REQUIRED
   track_type NMTOKEN #REQUIRED
   track_owner NMTOKEN #REQUIRED
```

Partner Code

The partner code is the trading partner code for track creator enterprise. This is only required if the sending enterprise is not the track creator.

```
<!ELEMENT partner_code (#PCDATA)>
```

The two track-by options are tracking by “O”ption and tracking by “S”tyle. This is the way the

sending enterprise would like to track the order by.

```
<!ELEMENT order (order_number, item_number, color_code?,
track_quantity?)>
<!ATTLIST order
   track_by NMTOKEN #REQUIRED
```

Order

The Order element is provided to allow updates to a WebTrack Order, as opposed to a Project. This element is required to identify the correct order to be updated.

```
<!ELEMENT order (order_number, item_number, color_code?,
track_quantity?)>
<!ATTLIST order
   track_by NMTOKEN #REQUIRED
```

Order Number

This is the order number in WebTrack If the track_type is “O”rder, then the order number must be present.

```
<!ELEMENT order_number (#PCDATA)>
```

Item Number

If the track_type is set to “O”rder and the track_by is set to “O”ption then the item_number must be populated. This number corresponds the style_id within WebTrack.

-

```
<!ELEMENT item_number (#PCDATA)>
```

Color Code

If the track_type is set to “O”rder and the track_by is set to “O”ption, then the color_code must be populated. This code corresponds to the color code within WebTrack Administration.

```
<!ELEMENT color_code (#PCDATA)>
```

Track Quantity

The user for the track-quantity is either the email address or user name of a WebTrack user who

Intends to make the update of track quantity. This user must be a valid user in the sending enterprise.

```
<!ELEMENT track_quantity (#PCDATA)>
<!ATTLIST track_quantity
   user CDATA #IMPLIED
>
```

Project

The project element is provided to allow updates to a WebTrack Project, as opposed to an Order.

```
<!ELEMENT project (project_name?, project_number, project_key?)>
```

Project Key

The project key is only relevant to styles being exported from Design to WebTrack. When a Style is exported to XML, each colour has this additional attribute key. This key's format is itemid-orderid, which is associated with the project. orderid is the internal id of the dummy order associated with the project. orderitemid is the internal id of the dummy order item in the dummy order associated with the project.

styleid is the internal id of the style. colourid is the internal id of the colour. In each case, internal id is the primary key.

> **Note:** The project key is only required if there are more than one project/event combination with the same project names.

```
<!ELEMENT project_key (#PCDATA)>
```

Project Name

Project Name is the name of the project in WebTrack. This entry is optional.

```
<!ELEMENT project_name (#PCDATA)>
```

Project Number

Project Number is the project number in WebTrack. If the track_type is “P”rojects, then the project number must be present.

```
<!ELEMENT project_number (#PCDATA)>
```

Parent Name/Child Name

Parent name is the name of the event being updated.

- If both parent_name and child_name are present, then the event appears in a child track and its name is specified by child_name.
- If parent_name is present and child_name is not present, then the event appears in a parent

  track and its name is specified by parent_name.
- If parent_name is not present and child_name is present, then the event could appear in a parent track or a child track and its name is specified by child_name. One restriction for this case is that if two events with the same name appear in a parent track and a child track respectively, then an error is reported.

--

```
<!ELEMENT parent_name (#PCDATA)>

<!ELEMENT child_name (#PCDATA)>
```

Date Type

Date-type specifies what kind of operation the sending enterprise would like to perform on the event. Valid values are “C” for complete, “R” for revised and “X” for cancel.

User is either the email address or user name of a WebTrack user who is intending to make the update of the event. This user must be a user in the sending enterprise.

Check specifies whether the sending enterprise would like WebTrack to checks the following:

- An event has been completed.
- The user who is intending to make the update of the event is the track owner.
- The user who is intending to make the update of the event is the event owner.
- An event to be completed is dependent on other events.
- An event has been suspended.
- Valid values are “Y” and “N”.

-->

```
<!ELEMENT dates (date)>
<!ATTLIST dates
   date_type NMTOKEN #REQUIRED
   user CDATA #IMPLIED
   check NMTOKEN #IMPLIED
>
```

Date is the date that the sending enterprise is updating as completed, revised, or cancelled.

-

```
<!ELEMENT date (#PCDATA)>
```

Diary Comments

The diary-comments user is the user that this dairy entry should be logged under. This user must be a valid user in the sending enterprise.

```
<!ELEMENT diary_comments (#PCDATA)>
<!ATTLIST diary_comments
   user CDATA #IMPLIED
>
```

A

Appendix: event.dtd

```
<?xml version="1.0" encoding="UTF-8"?>

<!ELEMENT events (event+)>
<!ATTLIST events
   version CDATA #IMPLIED
   enterprise CDATA #IMPLIED
>

<!ELEMENT event (partner_code?, order?, project?, parent_name?,
child_name?, dates?, diary_comments?)>
<!ATTLIST event
   action NMTOKEN #REQUIRED
   track_type NMTOKEN #REQUIRED
   track_owner NMTOKEN #REQUIRED
>

<!ELEMENT parent_name (#PCDATA)>

<!ELEMENT partner_code (#PCDATA)>

<!ELEMENT child_name (#PCDATA)>

<!ELEMENT order (order_number, item_number, color_code?,
track_quantity?)>
<!ATTLIST order
   track_by NMTOKEN #REQUIRED
>

<!ELEMENT item_number (#PCDATA)>

<!ELEMENT order_number (#PCDATA)>

<!ELEMENT color_code (#PCDATA)>

<!ELEMENT track_quantity (#PCDATA)>
<!ATTLIST track_quantity
   user CDATA #IMPLIED
>

<!ELEMENT project (project_name?, project_number, project_key?)>

<!ELEMENT project_key (#PCDATA)>

<!ELEMENT project_name (#PCDATA)>

<!ELEMENT project_number (#PCDATA)>

<!ELEMENT dates (date)>
<!ATTLIST dates
   date_type NMTOKEN #REQUIRED
   user CDATA #IMPLIED
   check NMTOKEN #IMPLIED
>

<!ELEMENT date (#PCDATA)>
```

```
<!ELEMENT diary_comments (#PCDATA)>
<!ATTLIST diary_comments
   user CDATA #IMPLIED
```

Example

```
<?xml version="1.0"?>
<events version="1.2" enterprise="100" xmlns:s="x-
schema:https://www.retail.com/import/schemas/eventsschema.xdr">
  <event action="U" track_owner="Y" track_type="P">
                <!-- Action is U for update, A for add, D for delete. You
currently cannot add or delete and event. -->
                <!-- Track owner is Y I have to be the owner, or N Has to be
partner ent -->
                <!-- Track type is P for project, O for order -->
    <project>
      <project_number>Ladies Skirts</project_number>
      <project_key>70949-153033-1472315773-Ladies Skirts/011</project_key>
    </project>
    <partner_code></partner_code>
    <parent_name>Spec Complete and Artwork Sent</parent_name>  <!-- Name of the
event -->

    <dates date_type="C" user="genericuser@retailer.com">
      <date>20051215000000</date>
    </dates>
    <diary_comments user="genericuser@retailer.com">This entry made by
genericuser</diary_comments>
  </event>
</events>
```