Oracle Taleo Enterprise Integration
Taleo Connect Export Selection Queries Specification

Part Number: E40110-02

February, 2015
Introduction

• Scope...........................................................................................................................2
Scope

This document covers the Selection Query data extraction language both at a high level and into the technical details. However, the use of such Selection Query expressions in the Integration Toolkit (ITK) export web service is not covered here. As such, examples will present the SQ-XML and not the required SOAP envelopes and ITK structure required from an actual invocation.

The new export service leverages the Akira project Selection Query framework, which uses the application meta model to express high level queries. The framework itself then translates this query into actual database SQL statements and handles paging and caching issues. Although SelectionQuery objects are actual constructs of the application server, it is possible to serialize them in XML format. It is precisely this XML format that is used in the Export web service request. The only difference is that the Integration Toolkit provides an entity/field mapping layer for isolation and versioning purposes.
Designing a Data Extraction Process

- Overview ..................................................................................................................................... 5
- Specific Considerations .............................................................................................................. 6
- Data Transformation .................................................................................................................. 7
- Choosing an Extraction Format ................................................................................................. 8
- Determining Application Model Mapping ................................................................................... 9
Overview

The first step in any design process is the requirement gathering. This is no different when building a data extraction process. However, there are a few specific information points to gather:

• The frequency of the extractions.
• The average or expected size of the extractions.
• The type of communication with the customer.
• The complexity of the mapping required between the Taleo and customer formats.
• The type of data required from the Taleo application and its relation to the actual model.
Specific Considerations

The main objective of the overall export environment is to extract the data quickly from the application to ensure maximum availability and responsiveness to end users, and execute the transformation and routing activities in the middleware or in the customer's own environment.
Data Transformation

One choice regarding the projection of the constructed SQ-XML is whether to use the report type format to execute part or the whole of the customer specific transformation or simply extract the raw data and transform it at a later stage. The recommendation here is to use the Selection Query for transformation only if:

- The transformation is simple (date formatting, concatenations, simple DECODE functions)
  AND
- The transformation can be completely expressed in SQ-XML (thus saving a customer specific integration package altogether).

If the second condition fails, then a customer specific package will have to exist anyway, and it is thus optimal to group all the transformation information in it. One immediate gain is that the SQ-XML document will be much lighter and easier to read and maintain. If the first condition fails, then performance will degrade when executing the requests on the application and database servers.
Choosing an Extraction Format

The export web service offers two distinct output format types: report- and entity-based. In addition, these types have actual physical representations: CSV and XML for report-type and T-XML for entity-type. Although discussed in more detail in the export web service documentation, it suffices to say here that data extracted using the entity-based format MUST match the application model structure and cannot show any extra or calculated fields. However, any extraction with multiple relations MAY gain from using the entity-based format, as report-type formats will duplicate information and multiply the number of records accordingly.

Once the report-type format has been chosen, then the CSV or XML format must be selected. The former is suited for situations where:

1. The extracted data can be transformed completely into the customer's specific format.
2. The size of the extracted data is very large (CSV is 50 to 75% smaller in size than XML).

The latter is best used when:

1. The extracted data must still be transformed (as better tools are available).
2. The size of the extracted data is manageable (say 10-30 MB).
Determining Application Model Mapping

Once the data to be extracted has been determined at a functional level, it must be expressed in terms of the application model. There are three tools available to do this that are described below.

Export Service Application Schema

Included within the WSDL file of the export web service or as a separate XSD file, the actual schema of the application model is the definitive reference that determines what data is exposed and thus possibly extracted. The schema defines the entities and for each an exhaustive list of fields and relationships. Among other things, it is based on this schema that you can determine a projection suitable for the entity-based format.

Export Service Mapping File

The link between the Integration Toolkit export service Selection Query statements and the application meta model is a mapping file that establishes a link between the XML tags (as seen in the schema) and the model specification for every exposed field and relation of every exposed entity. This mapping is actually used to generate the schema describing explicitly all the possible T-XML output representations. The mapping file is required when using the model documentation because the terms sometimes change slightly from the entity fields to the XML tags.

Application Model Documentation and Diagrams

Each application development team uses a modeling tool to build and implement the meta model. As such, a certain number of diagrams are available to describe the entities and their relationships. In certain cases, the various functional teams may document further the actual significance and use of the various fields in the entities.
Selection Query Format

- Overview.................................................................................................................... 11
- Basic Query............................................................................................................... 12
- Projections................................................................................................................ 13
- Filtering...................................................................................................................... 15
- Sorting........................................................................................................................ 18
- Ranking....................................................................................................................... 19
- Embedded Queries.................................................................................................... 20
- Sub-queries................................................................................................................ 22
- Advanced Topics........................................................................................................ 23
Overview

A meta model differs mainly from a relational data model in terms of the relationships created between its entities. As such, the Selection Query language differs from the SQL language mainly in the same manner. Since ultimately the Selection Query engine will translate all SQ-XML expressions into SQL statement to be executed against the physical model, Selection Query expressions are really very close to their SQL counterparts. Resources accustomed to creating SQL extraction scripts should easily grasp the workings of the Selection Query format. For SQL neophytes, the SQ-XML offers a simpler alternative for working with extraction instructions.

This section will present the "equivalent" SQL statement of the described SQ-XML documents. Please note that this is done ONLY for reference purposes. In almost all cases, the application entities and fields do NOT have the same name as the underlying physical elements. We purposely use the application model terms to clearly distinguish between examples and the SQL statements that would be actually generated by the export service.
Basic Query

An SQ-XML document typically starts with a query element. There are two required attributes to the query element: projectedClass and alias. The former represents the base entity from which the extraction will be built. The latter is an name unique throughout the expression that identifies the query.

A basic query starts out like this:

```xml
<query alias="BasicQuery" projectedClass="User"/>
```

**SQL equivalent**

```
FROM User
```
Projections

The first main elements of a query are the projections that represent what information is to be extracted for the selected entities. The projection elements are grouped in sequence within the projections element. This sequence is only relevant when using report-type formats. Entity-type formats always respect the application model schema, whatever the projection sequence.

The projection elements can be defined as any value understood by the Selection Query language.

The simplest case is to use a field of the projected class.

```xml
<query alias="SimpleProjection" projectedClass="User">
  <projections>
    <projection>
      <field path="UserName"/>
    </projection>
  </projections>
</query>
```

SQL equivalent

```
SELECT UserName FROM User
```

It is possible to assign an alias to a projection that serves two purposes. First, for the XML-type format, the alias will be shown in the output. Second, when a function is used, the Selection Query cannot deduce a default alias. Hence, it is required to explicitly specify one. Finally, when sub queries are involved, sometimes aliases are required to distinguish projections. This is the case because the default alias is the entity field name; so if both the main query and a sub-query project the Email field, then one of them will need an alias.

```xml
<query alias="ProjectionWithAlias" projectedClass="User">
  <projections>
    <projection alias="Login">
      <field path="UserName"/>
    </projection>
  </projections>
</query>
```

SQL equivalent

```
SELECT UserName AS Login FROM User
```

Another typical use case for projections is to apply a function to a given field. Most of the functions listed in the SQ schema respect the signatures of their SQL equivalents. In all cases, the schema indicates the number of parameters required for the function. Note that here an alias is required because of the use of a function.

```xml
<query alias="ProjectionWithFunction" projectedClass="User">
  <projections>
    <projection alias="Login">
      <toLowerCase>
        <field path="UserName"/>
      </toLowerCase>
    </projection>
  </projections>
</query>
```

SQL equivalent

```
SELECT LOWER(UserName) AS Login FROM User
```
The real strength of the Selection Query language comes from the application model relations. When such relations exist for the target data, then projecting it becomes straightforward. For example, by selecting the Department relation of the User entity (which is one-to-one according to the schema), you can access all of the Department entity fields. When the path expression only specifies the relation, then it is the entity key that is projected.

```xml
<query alias="ProjectionWithRelations" projectedClass="User">
    <projections>
        <projection>
            <field path="UserName"/>
        </projection>
        <projection>
            <field path="Department"/>
        </projection>
        <projection>
            <field path="Department.Name"/>
        </projection>
    </projections>
</query>
```

**SQL equivalent**

```
SELECT
    UserName, DepartmentNo, Department.Name
FROM
    User, Department
WHERE
    User.DepartmentNo = Department.No
```

When navigating a relation, you also have access to all the relations of the related entity. In the example below, since the Recruiter relation of the Department entity points to a User entity, then all the fields of that entity are again available.

```xml
<query alias="ProjectionWithDeepRelations" projectedClass="User">
    <projections>
        <projection>
            <field path="UserName"/>
        </projection>
        <projection>
            <field path="Department,Recruiter,UserName"/>
        </projection>
    </projections>
</query>
```

**SQL equivalent**

```
SELECT
    UserName, Recruiter.UserName
FROM
    User, Department, User Recruiter
WHERE
    User.DepartmentNo = Department.No AND
    Department.RecruiterNo = Recruiter.No
```
Filtering

The next query element includes the filters that represent what entities are to be selected. The filtering elements are grouped in sequence within the filterings element, although the sequence itself is not relevant.

The various filtering elements are implicitly linked by an AND logical operator.

The filtering elements can be defined as any filter understood by the Selection Query language, these are either logical operators or actual conditions. The simplest case is to use a standard equality condition. The equal is a binary operator and as such accepts two value child elements. One simple possibility is to use a field and a fixed value. We saw fields in the projection sections, the fixed values are of the normal types: numeric, string, etc.

```xml
<query alias="EqualityFilter" projectedClass="User">
  <projections>
    <projection>
      <field path="UserName"/>
    </projection>
  </projections>
  <filterings>
    <filtering>
      <equal>
        <field path="FirstName"/>
        <string>John</string>
      </equal>
    </filtering>
  </filterings>
</query>
```

**SQL equivalent**

```sql
SELECT UserName
FROM User
WHERE FirstName = 'John'
```

In the previous example, the SQ-XML is slightly more complex than the SQL equivalent. However, once again, the power of the expression language resides in the application model relations that allow a simple modification to filter on other relationships such as the Department name.

```xml
<query alias="RelationFilter" projectedClass="User">
  <projections>
    <projection>
      <field path="UserName"/>
    </projection>
  </projections>
  <filterings>
    <filtering>
      <equal>
        <field path="Department,Name"/>
        <string>Finance</string>
      </equal>
    </filtering>
  </filterings>
</query>
```

**SQL equivalent**

```sql
SELECT
  UserName
FROM
  User
WHERE Department,Name = 'Finance'
```
User, Department
WHERE
User.DepartmentNo = Department.No AND
Department.Name = 'Finance'

Applying a single logical condition can be done directly with the proper element.

<query alias="AndFilter" projectedClass="User">
  <projections>
    <field path="UserName"/>
  </projections>
  <filterings>
    <filtering>
      <and>
        <equal>
          <field path="FirstName"/>
          <string>John</string>
        </equal>
        <equal>
          <field path="LastName"/>
          <string>Doe</string>
        </equal>
      </and>
    </filtering>
  </filterings>
</query>

SQL equivalent

SELECT UserName
FROM User
WHERE
  FirstName = 'John' AND
  LastName = 'Doe'

However, applying several logical conditions must be done in an embedded manner, as most logical operator elements are binary (that is, accept only two child elements).

<query alias="MultipleAndFilters" projectedClass="User">
  <projections>
    <field path="UserName"/>
  </projections>
  <filterings>
    <filtering>
      <and>
        <and>
          <equal>
            <field path="FirstName"/>
            <string>John</string>
          </equal>
          <equal>
            <field path="LastName"/>
            <string>Doe</string>
          </equal>
        </and>
        <equal>
          <field path="MiddleInitial"/>
          <string>R</string>
        </equal>
      </and>
    </filtering>
  </filterings>
</query>
SQL equivalent

SELECT UserName
FROM User
WHERE
  FirstName = 'John' AND
  LastName = 'Doe' AND
  MiddleInitial = 'R'
### Sorting

The last main query element is the sorting instructions that represent in what order the selected entities will be shown. The sorting elements are grouped in sequence within the sortings element. The sequence determines what sorting instructions are applied first.

The sorting elements accept any value as child elements, but the main usage is with fields of the projected entity.

The sorting elements also accept an ascending attribute that determines the orientation of the particular ordering. Just like in SQL, this attribute defaults to true.

```xml
<query alias="Sorting" projectedClass="User">
  <projections>
    <projection>
      <field path="UserName"/>
    </projection>
  </projections>
  <sortings>
    <sorting>
      <field path="LastName"/>
    </sorting>
    <sorting ascending="false">
      <field path="FirstName"/>
    </sorting>
  </sortings>
</query>
```

**SQL equivalent**

```sql
SELECT UserName
FROM User
ORDER BY LastName ASC, FirstName DESC
```
Ranking

The ranking feature is not used in the context of data extraction.
Embedded Queries

The query element is also part of the any value collection, as such it can be used in most of the places described above.

Embedded queries can be used to project information that is not directly available through a relation. For example, each candidate answers questions when completing his/her profile, but let's assume the application model does not express this relationship. The following SQ-XML can then be used to retrieve the answer to a given question for a given user. The ownerQuery attribute of the field element allows embedded queries to reference information from the main (or any other) query.

```xml
<query alias="ProjectionEmbeddedQuery" projectedClass="Candidate">
  <projections>
    <projection>
      <field path="EmailAddress"/>
    </projection>
    <projection>
      <query alias="EmbeddedAnswerQuery" projectedClass="QuestionAnswer">
        <projections>
          <projection>
            <field path="Answer,Description"/>
          </projection>
        </projections>
        <filterings>
          <filtering>
            <and>
              <equal>
                <field path="Question">
                  <integer>-12</integer>
              </equal>
              <equal>
                <field path="Candidate"/>
                <field ownerQuery="ProjectionEmbeddedQuery" path="Candidate"/>
              </equal>
            </and>
          </filtering>
        </filterings>
      </query>
    </projection>
  </projections>
  <filterings>
    <filtering>
      <equal>
        <field path="EmailAddress">
          <string>johndoe@taleo.com</string>
      </equal>
    </filtering>
  </filterings>
</query>
```

**SQL equivalent**

```
SELECT 
  Candidate.EmailAddress, AnswerDescription.Description 
FROM 
  Candidate, QuestionAnswer, AnswerDescription 
WHERE 
  Candidate.EmailAddress = 'johndoe@taleo.com' AND
```
Embedded queries can also be used in filtering, where a typical case is to select a particular instance of an n-ary relation. Consider the case of candidates that may have several linked applications. The following SQ-XML will ensure only one requisition will be returned, linked to the latest application.

```xml
<query alias="FilteringEmbeddedQuery" projectedClass="Candidate">
  <projections>
    <projection>
      <field path="EmailAddress"/>
    </projection>
    <projection>
      <field path="Applications,Requisition,ContestNumber"/>
    </projection>
  </projections>
  <filterings>
    <filtering>
      <equal>
        <field path="EmailAddress"/>
        <string>johndoe@taleo.com</string>
      </equal>
    </filtering>
    <filtering>
      <equal>
        <field path="Applications,CreationDate"/>
        <query alias="EmbeddedApplicationQuery" projectedClass="PreselectionApplication">
          <projections>
            <projection alias="LatestDate">
              <maximum>
                <field path="CreationDate"/>
              </maximum>
            </projection>
          </projections>
          <filterings>
            <filtering>
              <equal>
                <field path="Candidate"/>
                <field ownerQuery="FilteringEmbeddedQuery" path="Number"/>
              </equal>
            </filtering>
          </filterings>
        </query>
      </equal>
    </filtering>
  </filterings>
</query>
```
Sub-queries

Sub-queries function much in the same manner as embedded queries although they are placed outside the main elements of a query. They can both reference and be referenced by the main query of the expression.

Consider the case where we would like to list all of the female candidates in the database and this information is available via the questions and answers entities. Suppose again that no explicit model relation exists between the candidate and his/her answers. We could first create a query that lists all candidate numbers for the given criteria, then the main query could use this list as its main filtering clause. In order to so, the sub-query must create a reference for the selected candidate numbers using the id attribute of the projection element. This value can then be used in the main query filtering by means of the ref attribute. The alias on the sub-query candidate number projection is required because the main query also uses the candidate number field and the generated SQL statement would have an ambiguous definition.

```xml
<query alias="SubQuery" projectedClass="Candidate">
  <subQueries>
    <query alias="AnswerSubQuery" projectedClass="QuestionAnswer">
      <projections>
        <projection id="FemaleCandidateNumber" alias="CandidateNo2">
          <field path="Candidate,No"/>
        </projection>
      </projections>
      <filterings>
        <filtering>
          <and>
            <equal>
              <field path="Question"/>
              <integer>-12</integer>
            </equal>
            <equal>
              <field path="Answer,Description"/>
              <string>Female</string>
            </equal>
          </and>
        </filtering>
      </filterings>
    </query>
  </subQueries>
  <projections>
    <projection>
      <field path="EmailAddress"/>
    </projection>
  </projections>
  <filterings>
    <filtering>
      <equal>
        <field path="No"/>
        <projection ref="FemaleCandidateNumber"/>
      </equal>
    </filtering>
  </filterings>
</query>
```
Advanced Topics

Locales

Locales are handled natively by the Selection Query framework and the meta layer. Each Selection Query is associated to a default locale, when none is specified the value 'en-US' is used. All multilingual fields are shown in the default locale. It is possible to override the query wide locale with the locale integration parameter (this is not in the SQ-XML as such, but really in the export service parameters).

For each field, the locale information can also be modified using the localeFiltering attribute. The default value of this attribute is defaultLocale, as described previously. Using the customLocales value enables the locales attributes, which accepts a comma-separated list of locales. The none value removes any filtering and will show the multilingual field in all available locales.

Custom Elements

The Selection Query language offers several SQL language elements, but does not assume to cover every possible case offered today. In particular, certain database systems might offer extra functionality that is not incorporated in the recognized standard. To accommodate these cases, certain custom elements are available.

The first and most common custom element is customFunction. This allows Selection Query expressions to go beyond the provided functions to transform the projected information. A common example in the current release is the TO_CHAR SQL function used to convert date information into a particular format. The customFunction element accepts any number of parameters, so it adapts to most if not all SQL supported functions.

```xml
<query alias="ProjectionWithCustomFunction" projectedClass="Candidate">
  <projections>
    <projection alias="Birthday">
      <customFunction name="TO_CHAR">
        <field path="Birthday"/>
        <string>MM-DD-YYYY</string>
      </customFunction>
    </projection>
  </projections>
</query>
```

SQL equivalent

```
SELECT TO_CHAR(BirthDay, 'MM-DD-YYYY') AS Birthday FROM Candidate
```

The other supported case in this release is the customValue element, albeit in a restricted manner. In order to compensate for certain technical limitations, sometimes the use of a custom value in an expression is required to access certain information. The customValue element accepts a single child expression element. For security reasons, only a TABLE.COLUMN value pattern is accepted.

References

References in SQ-XML function much in the same way as usual XML references. Most elements allow the use of an id attribute used to internally identify the construct. Once a definition is made with this attribute, other elements can use the ref attribute to specify the reference value. When using this attribute, all other attributes MUST be omitted (although this cannot strictly be enforced by the SQXML schema, it is noted as such).

Usually references are optional, allowing various constructs to be repeated in a convenient and concise manner. However, in the case of projections referring to fields in other queries or subqueries, the use of the id and ref attributes are mandatory (see example of this in the Sub-queries section).

User Defined Fields

User defined fields (UDFs) are handled natively by the Selection Query framework and the meta layer. In fact, from the Selection Query perspective, there are only a few distinctions between such fields and standard model fields.
When exported in the T-XML entity format, UDFs are grouped under the UDFs element with UDF children elements. In the case of a report-type extraction, there is no difference in the output.

In order to identify UDFs in projections or filter instructions, the code of the field must be used. Since such codes do not follow the same restrictions as actual entity fields (or database column names) character escaping is sometimes necessary. All non-alphanumeric characters must be replaced with their ASCII-code hexadecimal value prefixed with the underscore character. For example, if the UDF code is 'Extra Information_Shares-Options' (without the quotes), then the SQ-XML projection would be:

```xml
<projection>
  <field path="Extra_20Information_5fShares_2dOptions"/>
</projection>
```

Note: See the limitations section for the special case where UDF fields reference user defined selections (UDS).

**Attributes**

maxrecordcount

The maxrecordcount attribute sets the maximum number of records the export will return in sequence starting with row number one (1). The default value is minus one (-1) and exports all available rows.

```xml
  <soapenv:Header/>
  <soapenv:Body>
    <ns:findPartialEntities>
      <ns:query projectedClass="Location" alias="Location">
        <ns:projections>
          <ns:projection>
            <ns:field path="Code"/>
          </ns:projection>
        </ns:projections>
        <ns:filterings>
          <ns:filtering>
            <ns:equal>
              <ns:long>0</ns:long>
              <ns:long>1</ns:long>
            </ns:equal>
          </ns:filtering>
        </ns:filterings>
      </ns:query>
      <ns:attributes>
        <ns:entry>
          <ns:key>maxrecordcount</ns:key>
          <ns:value>2</ns:value>
        </ns:entry>
      </ns:attributes>
    </ns:findPartialEntities>
  </soapenv:Body>
</soapenv:Envelope>
```
Limitations and Known Issues
Limitations and Known Issues

Export UDS Value for a UDF

A technical issue prevents the direct extraction of the description of the selected value for UDFs based on user-defined selections. Normally, the Selection Query expression should be very simple for a candidate UDF field with a code of 'CertificationLevel'.

<projection>
    <field path="Candidate,CertificationLevel,Description"/>
</projection>

However, a more complex sub query is currently required that involves the use of a customValue element.

Exporting the Actual Locale in Report-type Mode

Because locales are handled natively in the meta layer (see the Advanced Topics section), the actual locale code is not an exposed field. When using the entity-type format, a multilingual field will have several value elements, each with a locale attribute. However, in a report-type format, although the actual translated value is shown, the corresponding language code is not.

A special work-around is available using a customValue element until an actual solution is found.

Exporting Data Stored in Optimization Tables

Several current customer data extractions utilize optimization database tables also used by our reporting solution. Because these are denormalized tables that have no functional significance, they are not modeled in the application meta layer. As such, the information regarding the OLF structures must be obtained through the normal functional relations.

Replacing Spaces in Projection Information

Currently, the replace element invoking the function of the same name cannot accept space characters. That is, the following expression will do nothing:

<query alias="LimitationsWithReplaceFunction" projectedClass="User">
    <projections>
        <projection>
            <replace>
                <field path="UserName"/>
                <string> </string>
                <string>_</string>
            </replace>
        </projection>
        </projections>
    </query>

Projecting all Fields of an Entity

There is no SQ-XML equivalent of the SQL projection wild-card (*). As such, if all exposed fields of a particular entity are to be exported, then they must all be explicitly listed in the projection collection.

Projecting all UDFs of an Entity

Because user-defined fields (UDFs) are considered just like normal fields in SQ-XML, the same remark as above also applies. In particular, relating to the ITK v.1.0 export service where the only option to extracting UDF data was to have all available UDF values exported. For the current version 2.0, UDF fields must be explicitly listed.
Projecting the Entity Number in Entity-based Format

A technical limitation of the SQ extraction engine prevents the internal reference number of the entity when using the entity-based T-XML format. Please note that this number is available when using the report-based CSV or XML formats.

Using Schema Validation on an Export

Currently there is an issue that prevents an export from being valid when compared to the export schema. The issue is caused by faulty ordering of some elements when serializing the xml. To circumvent this, a parameter was added to force the ordering of the elements when serializing. The new parameter is called ‘runtimefields.position’ and the possible values are ‘TOP’ (make the export valid against the schema) or ‘ORIGINAL’ (default behavior). To use this parameter in a synchronous webservice call you must add the following element to your soap request in the attributes section:

```xml
<soapenv:Body>
  <find:findPartialEntities>
    <find:query>
      ....
    </find:query>
    <find:attributes>
      <find:entry>
        <find:key>runtimefields.position</find:key>
        <find:value>top</find:value>
      </find:entry>
    </find:attributes>
  </find:findPartialEntities>
</soapenv:Body>
```

Eventually the parameter will go away and all exports will be valid by default when compared to their schema.