1. Overview

This note describes an approach to Registry ad hoc query support by providing a XML container based API. It attempts to meld together the discussions this week and provide a mutually acceptable framework to resolve the outstanding query implementation issues.

2 Definitions

This section defines some terminology that will be used in subsequent sections.

2.2 Focused Query

A focused query is one where the query interface is fixed to do a specific query task. A focused query approach pre-supposes what the client would wish to search for. Focused queries are defined statically by the Registry interface. Focused queries are analogous to calling statically defined methods that have a fixed signature. The current “browse and drill down” queries of the Registry are examples of focused queries.

2.3 Ad hoc Query API Components

The API seeks to extend the fixed focus query model and provide additional methods. As such it is not a generalized ad hoc query system nor a limited sub-set of some selected query syntax. Rather it provides a mechanism whereby Registry can offer extensible querying facilities based on business use case requirements.

These additional methods will be classified within the registry and therefore accessible via drill down access themselves. The focused query set therefore provides bootstrapping for registry ad hoc querying. This allows external users to discover exactly what ad hoc querying any given registry provides. To further support this, the ad hoc query API prescribes three categories of methods, *required*, *preferred*, and *specific*.

Required offer a suite of tools that the registry team has determined are logical a priori methods that will facilitate a broad range of business functional queries based on the RIM. They are closely analogous to the existing focused query model. They also serve to provide reference models for retrieving content based on the RIM to help guide implementers. These ad hoc methods differ from the focused query in that they seek to provide a single consist method metaphor and XML based implementation. While at the same time offering a more open value model typical of ad hoc querying, where the exact query content is provided by a XML structure (similar to that provided by the API of UDDI methods).

By contrast the focused query set is literally that, a set of interdependent methods. Ad hoc queries conversely can be used standalone, or in tandem with focused queries.

Preferred ad hoc queries are ones that are not critical to providing a functioning Registry implementation, and / or rely on technology specific techniques that not all Registry implementations may need or be capable of implementing. They have been determined to be broadly useful by the Registry team. *Specific* ad hoc queries are ones that local implementers have developed generic to a particular industry business need. An industry group would expect registries within that industry to support these. Conformance testing and industry certification can therefore be developed around the ad hoc API support model.

3 Long term strategy

The ad hoc API mechanism is designed to provide an initial acceptable system that can be widely implemented quickly and consistently. However the registry team is aware of the potential for an exponential explosion in ad hoc query methods, particularly in the *specific* category.

The overall problem appears to be somewhat intractable, since an open ad hoc query language is a holy grail not yet invented. Issues of security, behaviour and implementation consistency are apparent. Constrained subsets are a partial answer, but again there are major issues on what to constrain and how that affects implementation details and approaches.

Further complicating the issue is the fact that given the ebXML focus on XML, that an XML-centric query syntax would appear to be a natural preference. However XML query syntax is still an emerging technology that the W3C is yet to fully define.

From the business functional perspective the problem may not be as difficult as first appears. End users can be expected to require similar and related needs, and a constrained API can provide that. Certainly in a first phase, it is also appropriate to limit the functionality, to be more cautious, and so as to ensure a robust version 1.0 implementation. Also, this enables feedback from implementers and users to establish a true level of needed features over presumed features. Historically also, database access services have been provided restricted access, not open ad hoc querying externally.

Typical Registry query requirements search for content based on metadata submitted on the content as defined by Registry Information Model [RIM]. A query mechanism that supports the ability to search for content based on data that is part of submitted content is referred to as supporting content-based queries. Note that content-based queries are very similar in nature to totally ad

hoc queries. It should be noted that the information model fixes the Registry schema. As such totally ad hoc queries are not required for the registry.

Notice also, that the true issue is between registries. Locally a registry can implement its own ad hoc query interface to answer curious humans who wish to augment their searching experience. However the crux of the ad hoc API effects automated application-to-application (A2A) interactions to the registry, or multiple registries. These A2A queries are structured and driven by the actual schema semantics underpinning business processes. Given this constraint fixed queries make much more sense. An example would be a request for CPP information from an ebXML TRP layer that can be provided for neatly by providing structured methods that exactly match those requirements. Such requests are not subject to exponential explosions of complexity.

The ad hoc API approach therefore takes a sensible implementation strategy today that is historically a best choice, while providing for migration to a more extensive model in the future.

4 Implementation Details

This section presents details of the ad hoc API implementation.

4.1 Drill down discovery of Methods

The existing Registry drill down mechanisms will be used to implement this. The classification XML for this is:

```
<adhocAPImethods>
  <required>
    <method  name=”xxxxx” uri=”/yyyy/zzzz”/>
  </required>
  <preferred>
    <method  name=”aaaaa” uri=”/bbbb/cccc”/>
  </preferred>
  <specific>
    <method  name=”iiiiii” domain=”industryname” uri=”/jjjjj/kkkk”/>
 </specific>
</adhocAPImethods>
```

4.2 Mapping API Methods to RIM

The existing RIM defines a set of methods required to provide basic accessing of core information within the Registry itself. Each one of these will be provided with a method designation to allow provision of an ad hoc API method. Note that the AdHocQueryResponse will include a ManagedObjectList that will include heterogeneous elements (e.g. ExtrinsicObjects, Classification etc.) representing the classes specified in [RIM].

A non-exhaustive list includes the following *required* methods.

Simple Meta Data Based Queries

The simplest form of ad hoc queries is based upon metadata attributes specified for Registry content as specified in [RIM]. This form requires two parameters, an element name from the RIM, and a selection value to match. Variants can provide different matching criteria methods such as:

findMetadataItem.all ()
findMetadataItem.first()
findMetadataItem.all.containing()
findMetadataItem.all.containing.ignorecase()

Classification Queries

This section describes the various classification related queries that must be supported.

findClassificationItem.all ()
findClassificationItem.first()

Getting Objects Classified By a ClassificationNode

To get the collection of Objects classified by specified ClassificationNodes.

Getting ClassificationNodes That Classify an Object

To get the collection of Classification Nodes that classifies a specified Object.

Getting Root Classification Nodes

To get the collection of root Classification Nodes.

Getting Children of Specified Classification Node

To get the children of a specific Classification Node, given the ID reference to that node.

Getting Objects Classified By a Classification Node

To get the collection of Objects classified by specified Classification Nodes. Note that the query will also contain any objects that are classified by descendents of the specified Classification Nodes.

Association Queries

This section describes the various Association related queries.

Getting All Association Related to a Specified Object

To get the collection of Associations that has the specified Object as its source.

Getting Associations Based On Name, Role, Type

To get the collection of Associations that have specified Association attributes.

Getting Associated Objects Based On Association Attributes

To get the collection of Extrinsic Objects that are associated with a specified object, and based on an Association which is specified.

Package Queries

To find all packages that a specified object belongs to.

External Link Queries

These queries will find all External Links that a specified object is linked to.

Mapping of Predicates Involving Primitive Attributes

Most of the RIM interfaces methods are simple get/set methods that map directly to primitive attributes. For example the getName() and setName() methods on Object map to a name attribute of type String.

Mapping of Predicates Involving References

A few of the RIM interface methods return references to objects (e.g. Object#getAccessControlPolicy()). In such cases the references map to the ID attribute for the referenced object. This is again a special case of a primitive

attribute mapping. In this case the ID attribute is used as a reference pointer to the actual content item.

In summarizing the above methods, we can group them based on related RIM components.

RIM Procedures

Classification Related
- findClassifiedObjects
- findClassificationTree
- findRootClassificationNode

Association Related
- findAssociatedObjects
- findObjectsByExternalLink

Content Directed
- findObjectByName
- findObjectByContextPath
- findObjectByReferenceID
- findObjectByContent

Package Accessing
- findObjectsByPackage

Registry Deployment Support
- findRegistryFunctions
- findRegistryTRP

Business Based Functions

- findSupplierCPP
- findObjectsByOrganization
- findObjectsByIndustryClassification

The next section provides the actual XML to implement the above API methods with.

4.3 API structure for Methods

The objective is to provide a consistent XML API container that will always be interchanged between an application and a registry to perform the required ad hoc query request.

The container consists of the following structural components.

a) A binding for an optional registry object reference set that is to be provided (UDDI refers to this as a tModelBag).

b) A query method section that details the actual method to be invoked and the findQualifiers to be associated with the method.

c) An interchange control section that details how the method will interact and the response returned.

Therefore the actual XML structure for this is shown below.

Example 1.

```
<ebXMLadhocQuery>
  <requestItemBinding><RegistryItem/></requestItemBinding>

  <querySection method=”findObjectByContextPath”>
    <findQualifiers>
      <findQualifier>XMLstructure</findQualifier>
      <findQualifier>\root\some\items\content\</findQualifier>
      <findQualifier>parent</findQualifier>
    </findQualifiers>
  </querySection>

   <interactionControl action="returnURI"  maxRows="all" then="return" />

</ebXMLadhocQuery>
```

Example 2.

```
<ebXMLadhocQuery>
  <requestItemBinding/>

  <querySection method=”findObjectByName.contains.ignorecase”>
    <findQualifiers>
      <findQualifier>unitPrice</findQualifier>
    </findQualifiers>
  </querySection>

   <interactionControl action="returnGUID"  maxRows="all"  then="return"/>

</ebXMLadhocQuery>
```

Argument Definitions

- *requestItemBinding*: The registry object model contains specific reference elements and attributes that can be associated with a “fingerprint” of values for a particular reference item. A method may optionally require these in order to continue from a previously invoked query result set.

- *findQualifier*s: The collection of findQualifier elements can be used to alter the default behavior of the query method. (A value of %continues% indicates a cascading method call – see ‘then’ argument below).

- *maxRows*: This is an optional integer value, or the string “all”; allows the requesting program to limit the number of results returned.

- *action:* Specifies the mode that the method will use to return content to the calling program. Allowed modes are**: returnURI | returnContent | returnGUID | returnObject | returnRAW.**

- *then*: This is explicit control indicator. “Return” indicates that control will return immediately on completion of the method back to the caller. “Continue” indicates that the result set from this method be passed to the subsequent ebXMLadhocQuery as the findQualifiers block. This is useful to allow multiple related method calls to be passed at one time.

Returns:

The invoked method returns an XML result set on success. In the event that no matches were located for the specified criteria, the result set structure returned in the response the will be empty. In the event of a large number of matches, an *Operator Site* may truncate the result set. If this occurs, the response message will contain the *truncated* attribute with the value of this attribute set to *true*.

Caveats:

If any error occurs in processing this request, an ebXML error structure will be returned to the caller. The following error number information will be relevant:

Error_invalidKeyPassed: signifies that the *GUID_key* value passed did not match with any known key or key values. The error structure will signify which condition occurred first.

Error_tooManyOptions: signifies that more than one mutually exclusive argument was passed.

Error_unsupported: signifies that one of the findQualifier values passed was invalid.