OASIS

UDDI Spec TC

UDDI Version 3.0.2

UDDI Spec Technical Committee Draft, Dated 20041019

Document identifier:

uddi_v3

Current version:

http://uddi.org/pubs/uddi-v3.0.2-20041019.htm

Latest version:

http://uddi.org/pubs/uddi_v3.htm

Previous version:

http://uddi.org/pubs/uddi-v3.0.1-20031014.htm

Editors:

Luc Clement, Systinet
Andrew Hately, IBM
Claus von Riegen, SAP AG
Tony Rogers, Computer Associates

Contributors:

Tom Bellwood, IBM
Steve Capell
Luc Clement, Systinet
John Colgrave, IBM
Matthew J. Dovey
Daniel Feygin, UnitSpace
Andrew Hately, IBM
Rob Kochman, Microsoft
Paul Macias, LMI
Mirek Novotny, Systinet
Massimo Paolucci
Claus von Riegen, SAP AG
Tony Rogers, Computer Associates
Katia Sycara
Pete Wenzel, SeeBeyond Technology
Zhe Wu, Oracle

Abstract:

The UDDI Version 3.0.2 Specification describes the Web services, data structures and behaviors of all instances of a UDDI registry.

Status:

This specification has attained the status of Committee Draft. This document is updated periodically on no particular schedule.

Committee members should send comments on this Committee Specification to the uddi-spec@lists.oasis-open.org list. Others should subscribe to and send comments to the uddi-spec-comment@lists.oasis-open.org list. To subscribe, send an email message to uddi-spec-comment-request@lists.oasis-open.org with the word "subscribe" as the body of the message.

For information on whether any intellectual property claims have been disclosed that may be essential to implementing this Committee Specification, and any offers of licensing terms, please refer to the Intellectual Property Rights section of the UDDI Spec TC web page (http://www.oasis-open.org/committees/uddi-spec/ipr.php).

Copyrights:

Content

1 Introduction

Web services are meaningful only if potential users may find information sufficient to permit their execution. The focus of Universal Description Discovery & Integration (UDDI) is the definition of a set of services supporting the description and discovery of (1) businesses, organizations, and other Web services providers, (2) the Web services they make available, and (3) the technical interfaces which may be used to access those services. Based on a common set of industry standards, including HTTP, XML, XML Schema, and SOAP, UDDI provides an interoperable, foundational infrastructure for a Web services-based software environment for both publicly available services and services only exposed internally within an organization.

1.1 About this specification

This document describes the Web services and behaviors of all instances of a UDDI registry. Normative material is provided in the numbered chapters of the document and in the XML schemas which accompany this document. Supplementary non-normative commentary, explanations, and guidance may be found in the lettered appendices. In particular, first-time readers of this specification may find Appendix L *Glossary of Terms* useful.

This specification contains examples of XML data and URIs used in interacting with UDDI. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

The primary audiences for this document are:

- Programmers who want to write software that will directly interact with a UDDI registry.
- Programmers who want to implement a UDDI node
- Programmers who want to implement any of the Web services UDDI Nodes invoke

All implementations of the UDDI specification must provide support for the required Web services described here as well as the behaviors defined.

1.2 Language & Terms

RFC 2119: The keywords MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL, when they appear in this document, are to be interpreted as described in RFC 2119 found at http://www.faqs.org/rfcs/rfc2119.html.

MANDATORY, RECOMMENDED, and OPTIONAL: Beginning with this third version, the UDDI specification renders explicit which components of the UDDI specification are MANDATORY and MUST be implemented, which are RECOMMENDED and SHOULD be implemented, and which are OPTIONAL and MAY be implemented. It is important to note that OPTIONAL and RECOMMENDED elements of the specification, if they are implemented, MUST be implemented in the manner documented in this specification.

Separation of operational issues: In this third version of the UDDI Specification the trend begun in Version 2 to separate normative behavior from UDDI registry and node policy is completed. For instance, authorization has been called out as a policy decision. A similar separation of normative behavior and registry content has also been carried out. For example, the requirement to support specific category systems has been removed from this version of the specification.

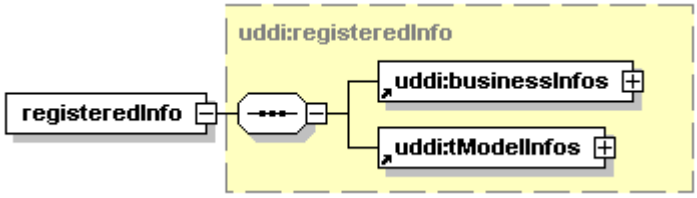
1.3 Diagrams Used in this document

1.3.1 Attributes and elements

UDDI uses the XML Schema Language (See http://www.w3.org/TR/xmlschema-0/, http://www.w3.org/TR/xmlschema-1/ and http://www.w3.org/TR/xmlschema-2/) and its terminology, such as "sequence" and "choice" to formally describe its data structures. The diagrams[1] used in this specification show the structure and cardinality of the elements used in these structures. Attributes are not shown in the diagrams, but explained in the corresponding documentation.

1.3.2 Element structure

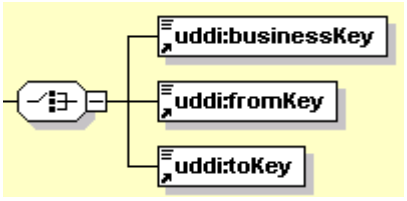
1.3.2.1 Sequence



The octagonal symbol with the horizontal "dotted" line indicates "sequence of." This diagram says the element *registeredInfo* consists of elements *businessInfos* and *tModelInfos*. All three elements are defined in the namespace whose prefix is "uddi".

The fact that *businessInfos* and *tModelInfos* have a box with a "+" in it at their right-hand end indicates that there is more structure to them than is shown in the diagram.

1.3.2.2 Choice



The switch-like symbol indicates a choice. In this case, a choice between the elements *businessKey, fromKey,* and *toKey*.

None of these has more structure than is given in the diagram (there are no boxes with a "+" in them at their right-hand ends). That they are adorned with a small series of horizontal lines in their upper left corners indicates that each is a non-empty element.

1.3.3 Cardinality

1.3.3.1 Optional, one

[1] Diagrams provided in this specification were produced by the ©XML Spy editor, Altova GmbH and Altova, Inc.

The dashed line indicates that the element *listDescription* is optional. The fact that it is not adorned with some other cardinality indicator (see below) says there can be at most one of them.

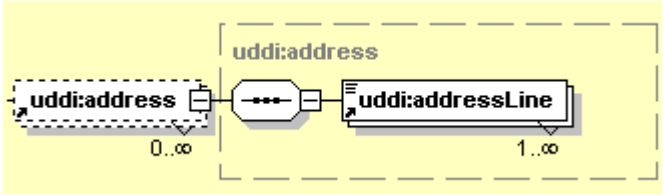
1.3.3.2 Mandatory, one

There must be exactly one of the element *businessKey*.

1.3.3.3 Optional, repeating

The element *assertionStatusItem* is optional and may appear an indeterminate number of times. The number of times it may appear is given by the adornment "0..∞", a cardinality indicator meaning "zero to infinity". Other numbers may appear to indicate different cardinalities.

1.3.3.4 Mandatory, repeating



The element *addressLine* must appear at least once and may appear an indeterminate number of times.

1.4 Related Documents

1.4.1 Translations of the UDDI Specification

Translations of the UDDI Specifications may be produced, by the UDDI specification technical committee of OASIS, or by others. In all instances the English version of the document is the official version; in case of discrepancy the English version shall be the definitive source.

1.4.2 Best Practices and Technical Notes

To provide guidance on the use of UDDI registries, the UDDI specification technical committee of OASIS from time to time publishes "Best Practices" and "Technical Notes". The contents of these documents are not a part of this specification. See http://www.oasis-open.org/committees/uddi-spec/doc/bps.htm for further information on Best Practices and http://www.oasis-open.org/committees/uddi-spec/doc/tns.htm for information on Technical Notes.

1.5 Base UDDI Architecture

1.5.1 UDDI Data

This specification presents an information model composed of instances of persistent data structures called entities. Entities are expressed in XML and are persistently stored by UDDI

nodes. Each entity has the type of its outer-most XML element. A UDDI information model is composed of instances of the following entity types:

- businessEntity: Describes a business or other organization that typically provides Web services.
- businessService: Describes a collection of related Web services offered by an organization described by a businessEntity.
- bindingTemplate: Describes the technical information necessary to use a particular Web service.
- tModel: Describes a "technical model" representing a reusable concept, such as a Web service type, a protocol used by Web services, or a category system.
- publisherAssertion: Describes, in the view of one businessEntity, the relationship that the businessEntity has with another businessEntity.[2]
- subscription: Describes a standing request to keep track of changes to the entities described by the subscription.

1.5.2 UDDI Services and API Sets

This specification presents APIs that standardize behavior and communication with and between implementations of UDDI for the purposes of manipulating UDDI data stored within those implementations. The API's are grouped into the following API sets.

1.5.2.1 Node API Sets

- UDDI Inquiry
- UDDI Publication
- UDDI Security
- UDDI Custody Transfer
- UDDI Subscription
- UDDI Replication

1.5.2.2 Client API Sets

- UDDI Subscription Listener
- UDDI Value Set

1.5.3 UDDI Nodes

A set of Web services supporting at least one of the Node API sets is referred to as a UDDI node. A UDDI node has these defining characteristics:

1. A UDDI node supports interaction with UDDI data through one or more UDDI API sets
2. A UDDI node is a member of exactly one UDDI registry.
3. A UDDI node conceptually has access to and manipulates a complete logical copy of the UDDI data managed by the registry of which it is a part. Moreover, it is this data

[2] If identical publisherAssertions are made from the views of both businessEntities, a "relationship" is formed between them. See Section 3.7 publisherAssertion.

which is manipulated by any query and publish APIs supported by the node. Typically, UDDI replication occurs between UDDI nodes which reside on different systems in order to manifest this logical copy in the node.

The physical realization of a UDDI node is not mandated by this specification.

1.5.4 UDDI Registries

One or more UDDI nodes may be combined to form a UDDI Registry. The nodes in a UDDI registry collectively manage a particular set of UDDI data. This data is distinguished by the visible behavior associated with the entities contained in it.

A UDDI Registry has these defining characteristics.

1. A registry is comprised of one or more UDDI nodes.
2. The nodes of a registry collectively manage a well-defined set of UDDI data. Typically, this is supported by the use of UDDI replication between the nodes in the registry which reside on different systems.
3. A registry MUST make a policy decision for each policy decision point. It MAY choose to delegate policy decisions to nodes. See Chapter 9 *Policy* for details.

The physical realization of a UDDI Registry is not mandated by this specification.

1.5.5 Affiliations of Registries

The entities businessEntity, businessService, bindingTemplate, tModel form the core data structures of UDDI. Within a registry, each instance of the core data structures is uniquely identified by a UDDI key. By choosing appropriate policies, multiple registries may form a group, known as an "affiliation", whose purpose is to permit controlled copying of core data structures among them. A UDDI registry affiliation has these defining characteristics.

1. The registries share a common namespace for entity keys.
2. The registries have compatible policies for assigning keys to entities.
3. The policies of the registries permit publishers to assign keys

1.5.6 Person, Publisher and Owner

When publishing information in a UDDI registry the information becomes part of the published content of the registry. During publication of an item of UDDI information, a relationship is established between the publisher, the item published and the node at which the publish operation takes place. The glossary contains definitions of the terms person, publisher and owner.

This specification defines a relationship between these three terms and leaves the binding of these abstract relationships to be determined by the policies of the registry and its nodes at implementation. It is important to review Chapter 9 on policy to understand how different implementations can define different policies but remain consistent with the UDDI specification.

1.5.7 Transfer of ownership

As the owner of datum, a person can initiate the transfer of ownership of the datum to another publisher within the registry. Section 5.4 *Custody and Ownership Transfer API* describes the transfer of ownership within UDDI.

1.5.8 Data Custody

Generally speaking, data is replicated between nodes of a UDDI registry using a replication protocol. Registries that choose to use the replication protocol defined in Section 7.4

Replication API Set MUST enforce the following data custody policy. (Registries which choose otherwise incur no such requirement.)

Each node has custody of a portion of the aggregate data managed by the registry of which it is a part. Each datum is by definition in the custody of exactly one such node. A datum in this context can be a businessEntity, a businessService, a bindingTemplate, a tModel, or a publisherAssertion. Changes to a datum in the registry MUST originate at the node which is the custodian of the datum. The registry defines the policy for data custody and, if allowed, the custodian node for a given datum can be changed; such custody transfer processes are discussed in Section 5.4 *Custody and Ownership Transfer API*.

1.6 Representing Information within UDDI

For Web services to be meaningful there is a need to provide information about them beyond the technical specifications of the service itself. Central to UDDI's purpose is the representation of data and metadata about Web services. A UDDI registry, either for use in the public domain or behind the firewall, offers a standard mechanism to classify, catalog and manage Web services, so that they can be discovered and consumed. Whether for the purpose of electronic commerce or alternate purposes, businesses and providers can use UDDI to represent information about Web services in a standard way such that queries can then be issued to a UDDI Registry – at design-time or run-time – that address the following scenarios:

- Find Web services implementations that are based on a common abstract interface definition.
- Find Web services providers that are classified according to a known classification scheme or identifier system.
- Determine the security and transport protocols supported by a given Web service.
- Issue a search for services based on a general keyword.
- Cache the technical information about a Web service and then update that information at run-time.

These scenarios and many more are enabled by the combination of the UDDI information model and the UDDI API set. Because the information model is extremely normalized, it can accommodate many different types of models, scenarios and technologies. The specification has been written to be flexible so that it can absorb a diverse set of services and not be tied to any one particular technology. While a UDDI Node exposes its information as an XML Web service, it does not restrict the technologies of the services about which it stores information or the ways in which that information is decorated with metadata.

1.6.1 Representing Businesses and Providers with "businessEntity"

One top-level data structure within UDDI is the businessEntity structure, used to represent businesses and providers within UDDI. It contains descriptive information about the business or provider and about the services it offers. This would include information such as names and descriptions in multiple languages, contact information and classification information. Service descriptions and technical information are expressed within a businessEntity by contained businessService and bindingTemplate structures.

While the name of XML entity itself has the word *business* embedded in it, the structure can be used to model more than simply a "business" in its common usage. As the top-level entity, *businessEntity* can be used to model any "parent" service provider, such as a department, an application or even a server. Depending on the context of the data in the entire registry, the appropriate modeling decisions to represent different service providers can vary.

1.6.2 Representing Services with "businessService"

Each businessService structure represents a logical grouping of Web services. At the service level, there is still no technical information provided about those services; rather, this structure allows the ability to assemble a set of services under a common rubric. Each businessService is the logical child of a single businessEntity. Each businessService contains descriptive information – again, names, descriptions and classification information -- outlining the purpose of the individual Web services found within it. For example, a businessService structure could contain a set of Purchase Order Web services (submission, confirmation and notification) that are provided by a business.

Similar to the businessEntity structure, the term business is embedded within the name *businessService*. However, a suite of services need not be tied to a business per se, but can rather be associated with a provider of services, given a modeling scenario that is not based on a business use case.

1.6.3 Representing Web services with "bindingTemplate"

Each bindingTemplate structure represents an individual Web service. In contrast with the businessService and businessEntity structures, which are oriented toward auxiliary information about providers and services, a bindingTemplate provides the technical information needed by applications to bind and interact with the Web service being described. It must contain either the access point for a given service or an indirection mechanism that will lead one to the access point.

Each binding Template is the child of a single businessService. The containing parents, a bindingTemplate can be decorated with metadata that enable the discovery of that bindingTemplate, given a set of parameters and criteria.

1.6.4 Technical Models (tModels)

Technical Models, or tModels for short, are used in UDDI to represent unique concepts or constructs. They provide a structure that allows re-use and, thus, standardization within a software framework. The UDDI information model is based on this notion of shared specifications and uses tModels to engender this behavior. For this reason, tModels exist outside the parent-child containment relationships between the businessEntity, businessService and bindingTemplate structures.

Each distinct specification, transport, protocol or namespace is represented by a tModel. Examples of tModels that enable the interoperability of Web services include those based on Web Service Description Language (WSDL), XML Schema Definition (XSD), and other documents that outline and specify the contract and behavior – i.e., the interface – that a Web Service may choose to comply with. To describe a Web service that conforms to a particular set of specifications, transports, and protocols, references to the tModels that represent these concepts are placed in the bindingTemplate. In such a way, tModels can be re-used by multiple bindingTemplates. The bindingTemplates that refer to precisely the same set of tModels are said to have the same "technical fingerprint" and are of the same type. In this way, tModels can be used to promote the interoperability between software systems.

It is important to note that such technical documents and the supporting documentation necessary to a developer using Web services are not stored within the UDDI registry itself. A UDDI tModel simply contains the addresses where those documents can be found. A tModel, however, contains more than just URLs; it also stores metadata about the technical documents and an entity key that identifies that tModel.

Because tModels can represent any unique concept or construct, they have usage beyond the software interoperability scenario described above. They can also be used to represent other concepts within the UDDI information model, such that metadata concepts are reused

throughout the model. For example, tModels are used for the following other purposes within UDDI:

- Transport and protocol definitions such as HTTP and SMTP. (See below and also Section 11.1.1 *uddi-org:types* for a description.)
- Value sets including identifier systems, categorization systems and namespaces. (See Section 3.3 *businessEntity Structure* and Appendix F *Using Categorization* for a description of how value sets are used in UDDI.)
- Structured categorizations using multiple value sets called "categorization groups."
- Postal address formats. (See Section 3.3.2.7 *address* and Appendix B *Internationalization* for a description.)
- Find qualifiers used to modify the behavior of the UDDI find_xx APIs. (See Section 5.1.4 *findQualifiers* for a description.)
- Use type attributes that specify the kind of resource being referred to by a URI reference. (See, for example, Section 3.5.2.1 *accessPoint*.)

The use of tModels is essential to how UDDI represents data and metadata. The UDDI specification defines a set of common tModels that can be used canonically to model information within UDDI. If a concept that is required to model a particular scenario does not exist in a registry, a user should introduce that concept by saving a tModel containing the URL of the relevant overview documents.

1.6.5 Taxonomic Classification of the UDDI entities

Data is worthless if it is lost within a mass of other data and cannot be distinguished or discovered. If a client of UDDI cannot effectively find information within a registry, the purpose of UDDI is considerably compromised. Providing the structure and modeling tools to address this problem is at the heart of UDDI's design. The reification of data within UDDI is core to its mission of description, discovery and integration. It achieves this by several means.

First, it allows users to define multiple taxonomies that can be used in UDDI. In such a way, multiple classification schemes can be overlaid on a single UDDI entity. This capability allows organizations to extend the set of such systems UDDI registries support. One is not tied to a single system, but can rather employ several different classification systems simultaneously.

Second, UDDI allows such classification systems to be used on every entity within the information model. It defines a consistent way for a publisher to add any number of classifications to their registrations. It is important that taxonomies are used when publishing data into a UDDI registry. Whether standard codes are used (such as the United Nations Standard Products and Services Code System (UNSPSC)) or a new taxonomy is created and distributed, it is imperative that UDDI data -- businessEntity, businessService, bindingTemplate and tModel elements alike – are attributed with metadata.

Third, the UDDI Inquiry API set provides the ability to issue precise searches based on the different classification schemes. A range of queries that perform different joins across the UDDI entities can be generated, such that data can be discovered and accessed. Also, registering information such as industry codes, product codes, geography codes and business identification codes allows other search services to use this classification information as a starting point to provide added-value indexing and classification.

Classification and identification systems, taken together, are called "value sets" in UDDI. Value sets may be "checked" or "unchecked". Both checked and unchecked value sets are used for categorization and identification. The difference between them is that whenever a checked value set is used, the use is inspected to see that it conforms to the requirements of the value set. Unchecked value sets do not have their uses checked.

1.7 Introduction to Security

The security model for a UDDI registry can be characterized by the collection of registry and node policies and the implementation of these policies by a UDDI node. This specification details a list of policies that MUST be defined by registries and nodes in Chapter 9 *Policy.* This specification also describes how policies SHOULD be modeled.

Several optional and extensible mechanisms for implementing nodes, registries and clients with a particular security model are described in this specification. The principal areas of security policies and mechanisms in the UDDI specification are related to data management, user identification, user authentication, user authorization, confidentiality of messages and integrity of data.

In order to authorize or restrict access to data in a UDDI registry, an implementation of a UDDI node MAY be integrated with one or more identification systems. An implementation specific policy MUST identify the identification system(s) used. Integration of UDDI APIs and data with an identification system MAY be implemented through the authentication and authorization APIs to provide access control as described in Section 5.3 *Security Policy API Set*. Other authentication and authorization mechanisms and policies are represented in UDDI through use of tModels to describe the Web services of a UDDI node.

UDDI also supports XML Digital Signatures on UDDI data to enable inquirers to verify the integrity of the data with respect to the publisher.

The security model for a registry and node can be extended beyond the mechanisms described in this specification and represented by modeling the UDDI Web services and through node and registry policy documentation.

1.8 Introduction to Internationalization

As part of its aim of providing a registry for *universal* description, discovery and integration, the UDDI specification includes support for internationalization features. These features fall into two broad groups:

- Support for multi-regional businesses, organization, and other Web service providers to:
  - Describe their operations across international or inter-region units
  - Specify the timezone of each operation's contacts
- Support for internationalization of UDDI data and services such as:
  - XML and the Unicode Character Set
  - Postal address
  - Use of multiple languages or multiple scripts of the same language
  - Mechanisms to specify additional language-specific sorting order
  - Consistent search results independent of language of information being searched

1.8.1 Multi-regional businesses

The UDDI specification provides features that enable Web service providers to describe the location of different aspects of the business, e.g. where it offers its products and services, where it is located, or even where it has stores, warehouses, or other branches.

1.8.2 XML and Unicode Character Set

The UDDI specification uses XML and the Unicode Character Set (up to and including version 3.0 of the Unicode Standard). By basing the programming interface on XML, multilingual handling capability is automatically achieved as XML uses the Universal Character Set (UCS) defined by both the Unicode Consortium and ISO 10646. The UCS is a character set that encompasses most of the language scripts used in computing.

1.8.3 Standardized Postal Address

In UDDI, an <address> element consists of a list of <addressLine> elements. While this is useful for publishing addresses in a UDDI registry or simply printing them on paper, the address' logical structure and meaning is not explicit.

Moreover, different geographical regions specify their postal addresses differently

- By having different subelements (e.g. subdivisions, suburbs, lots, building identifications, floor numbers)
- By grouping/sequencing the subelements.

To overcome the first concern, the UDDI specification exposes an address' structure and meaning by the use of attributes within each <addressLine> element to specify that line's structure and meaning.

To overcome the second concern, the UDDI Business Registry has specified a canonical postal address structure with common address subelements (e.g. states, cities). This canonical address structure describes address data via name/code pairs, enabling each common address subelement to be identified by name or code[3].

1.8.4 Use of Multi-languages and Multi-scripts

Multinational businesses or businesses involved in international trading at times require the use of possibly several languages or multiple scripts of the same language for describing their business. The UDDI specification supports this requirement through two means, first by specifying the use of XML with its underlying Unicode representation, and second by permitting the use of the xml:lang attribute for various items such as names, addresses, and document descriptions to designate the language in which they are expressed. Further information on this may be found in Section 3.3.2.3 *name*.

1.8.5 Adding Language-specific Sort Orders

The Universal Character Set supported through XML consists of characters of most of the language scripts of the world. Each character has a distinct collation weight within the language for use in the collation sequencing process. Handling the sort orders of different language scripts, i.e. the assignment of collation weight values, can be very different, with the complexity of handling dependent on whether the script is alphabetic, syllabic, or ideographic. Some examples of sort order handling issues are:

- Where multiple languages share the same alphabetic script, it is possible for a common character to have different collation weights when used in the different languages.
- Ideographic languages have large character repertoires with multiple collation sequencing possibilities depending on whether phonetic or stroke-order sequencing is chosen.

[3] See Section 3.3.3.65 address and Appendix D Internationalization.

- Where languages are bicameral (having upper and lower cases), collation sequencing could depend on whether case-sensitive or insensitive sorting is required. Conversely, specifying case-sensitive sort for non-bicameral languages is meaningless.
- Where the language inherently has an obvious collation sequence, fastest sorting is achieved by using binary sort.

The UDDI specification allows the collation sequence of results returned by the APIs to be specified via qualifiers. The specification also supports a mechanism to specify additional language-specific collation sequences for collating returned results.

1.8.6 Consistent Internationalized Search

The existence within the Universal Character Set of combining characters and of multiple representations for what users perceive as the same character results in different (by content and sometimes by length as well) XML strings that are the same when rendered visually. These different XML strings, though different in their encoded binary form, should produce positive match results during any search operation. This requirement makes it necessary to define a canonical XML string representation. The canonical representation chosen is that of the Unicode Normalization Form C[4]. For further details, see Section 4.6.1.1 *Normalization and Canonicalization*.

[4] Specified in Unicode Technical Standard, Technical Report #15 available at http://www.unicode.org/unicode/reports/tr15/

2 UDDI Schemas

UDDI uses the XML Schema Language (See http://www.w3.org/TR/xmlschema-0/, http://www.w3.org/TR/xmlschema-1/ and http://www.w3.org/TR/xmlschema-2/) to formally describe its data structures. A UDDI node MUST use an XML processor that meets the definition of a minimally conforming schema aware processor as defined in XML Schema Part 1: Structures. The XML processor must further understand the references to schema components (see Section 4.2.3 of XML Schema Part 1: Structures) across namespaces which result from the import statements in the UDDI schemas. The complete definition comprises 9 schema files, as described below.

UDDI API Schema	
Schema file	http://uddi.org/schema/uddi_v3.xsd
Target namespace	urn:uddi-org:api_v3
Referenced/imported namespaces	http://www.w3.org/2001/XMLSchema http://www.w3.org/2000/09/xmldsig# http://www.w3.org/XML/1998/namespace
Description	This is the main UDDI Schema file. It defines all of the common UDDI data types and elements as well as those used in the Inquiry, Publishing, and Security API sets.

UDDI Custody Schema	
Schema file	http://uddi.org/schema/uddi_v3custody.xsd
Target namespace	urn:uddi-org:custody_v3
Referenced/imported namespaces	urn:uddi-org:api_v3 urn:uddi-org:repl_v3 http://www.w3.org/2001/XMLSchema
Description	This is the schema for the UDDI Custody and Ownership Transfer API set.

UDDI Subscription Schema	
Schema file	http://uddi.org/schema/uddi_v3subscription.xsd
Target namespace	urn:uddi-org:sub_v3
Referenced/imported namespaces	urn:uddi-org:api_v3 http://www.w3.org/2001/XMLSchema
Description	This is the schema for the UDDI Subscription API set.

UDDI Subscription Listener Schema	
Schema file	http://uddi.org/schema/uddi_v3subscriptionListener.xsd
Target namespace	urn:uddi-org:subr_v3
Referenced/imported namespaces	urn:uddi-org:api_v3 urn:uddi-org:sub_v3 http://www.w3.org/2001/XMLSchema
Description	This is the schema for the UDDI Subscription Listener API set.

UDDI Replication Schema	
Schema file	http://uddi.org/schema/uddi_v3replication.xsd
Target namespace	urn:uddi-org:repl_v3
Referenced/imported namespaces	urn:uddi-org:api_v3 http://www.w3.org/2000/09/xmldsig# http://www.w3.org/2001/XMLSchema
Description	This is the schema for the UDDI Replication API set.

UDDI Value Set Validation Schema	
Schema file	http://uddi.org/schema/uddi_v3valueset.xsd
Target namespace	urn:uddi-org:vs_v3
Referenced/imported namespaces	urn:uddi-org:api_v3 http://www.w3.org/2001/XMLSchema
Description	This is the schema for the UDDI Value Set Validation API set.

UDDI Value Set Caching	
Schema file	http://uddi.org/schema/uddi_v3valuesetcaching.xsd
Target namespace	urn:uddi-org:vscache_v3
Referenced/imported namespaces	urn:uddi-org:api_v3 http://www.w3.org/2001/XMLSchema
Description	This is the schema for the UDDI Value Set Data API set.

UDDI Policy	
Schema file	http://uddi.org/schema/uddi_v3policy.xsd
Target namespace	urn:uddi-org:policy_v3
Referenced/imported namespaces	http://www.w3.org/2001/XMLSchema http://www.w3.org/2000/09/xmldsig# http://www.w3.org/XML/1998/namespace
Description	This is the schema for the UDDI Policy Document for the Policy Service.

UDDI Policy Instance Parameters	
Schema file	http://uddi.org/schema/uddi_v3policy_instanceParms.xsd
Target namespace	urn:uddi-org:policy_instanceParms_v3
Referenced/imported namespaces	http://www.w3.org/2001/XMLSchema
Description	This is the schema for the instance parameters that are used in modeling UDDI policies.

2.1 Schema versioning

UDDI follows the commonly encountered convention of changing the target namespace whenever a specification revision changes the schema in a way that changes the set of documents that is valid under the schema. In addition, UDDI changes the target namespace whenever a specification revision changes in a way that changes the behavior a compliant registry is permitted to display with respect to the schema, even if the set of documents that are valid under the schema remains unchanged. UDDI does not change the target namespace for other kinds of changes. For example, the target namespace is not changed for purely editorial or formatting errata, either to the Specification or to a schema.

The form of the target namespace is (using ABNF notation):

```
namespace = "urn:uddi-org:" schemaName  "_v" versionNumber [":" revisionNumber]
versionNumber = decimalInteger
revisionNumber = decimalInteger
schemaName = "api" / "custody" / "sub" / "subr" / "repl" / "vs" / "vscache" / "policy" /
"policy_instanceParms"
decimalInteger = Unsigned integer with no leading zeroes.
```

Where versionNumber is the same as the version number of UDDI of which the schema is a part. E.g., for UDDI v3, versionNumber is 3. The value of revisionsNumber is the number of the revision to the specification in which the schema is used.

When the specification is first released revisionsNumber is 0. It is incremented by 1 with each released revision.

So, for example, namespace for the UDDI API Schema corresponding to UDDI v3 in its first release is "urn:uddi-org:api_v3:0".

In addition, the UDDI schemas use the version attribute of the schema element to mark changes to the text of the schema in the following manner. The value of the version attribute is an unsigned decimal integer. When a schema is first created for a given version of UDDI its

version is 0. The value of version is incremented by at least 1 each time the schema is made publicly available.

2.2 Schema Extensibility

As defined in the UDDI schemas, all UDDI data structures are designed to permit UDDI node implementers to extend them using the XML Schema derivation-by-extension feature. While extending the UDDI schemas in this way can be a relatively straightforward process, designing an extension that includes behavioral modification is likely to be a complex undertaking that should be done with considerable care. See Appendix H *Extensibility* for more information on extending UDDI.

2.3 Element and attribute types and lengths

To ease the replication of data between nodes of a registry and to facilitate sharing data among the registries of an affiliation, UDDI imposes length restrictions on the types in its information model. The following tables summarize all the stored elements and attributes in the UDDI schemas that correspond to XML schema simpleTypes. They provide data types and, for those whose length is not specified by XML, their allowed lengths. The lengths are the storage length limits for information that is saved in a UDDI registry, given in Unicode characters. Since these limits are imposed in the schemas, structures containing data that exceeds the constraints depicted below are not valid. The lengths specified in the UDDI schemas are the definitive source for type and length information.

2.3.1 Data structure, publication API, inquiry API and security API

Element/attribute Name	Data Type	Length
accessPoint	string	4096
addressLine	string	80
authInfo	string	4096
bindingKey	anyURI	255
businessKey	anyURI	255
deleted	boolean	
description	string	255
discoveryURL	anyURI	4096
email	string	255
fromKey	anyURI	255
instanceParms	string	8192
keyName	string	255
keyValue	string	255
name	string	255
operator	string	255
overviewURL	anyURI	4096
personName	string	255
phone	string	50
serviceKey	anyURI	255

Element/attribute Name	Data Type	Length
sortCode	string	10
tModelKey	anyURI	255
toKey	anyURI	255
useType	string	255
completionStatus	NMTOKEN	32
xml:lang	string	26

2.3.2 Subscription API

Element/attribute Name	Data Type	Length
brief	boolean	
endPoint	dateTime	
notificationInterval	duration	
expiresAfter	dateTime	
startPoint	dateTime	
maxEntities	integer	
subscriptionKey	anyURI	255

2.3.3 Replication API

Element/attribute Name	Data Type	Length
acknowledgementRequested	boolean	
nodeId	anyURI	255
notifyingNode	anyURI	255
originatingUSN	integer	
operatorNodeID	anyURI	255
requestingNode	anyURI	255
responseLimitCount	integer	

3 UDDI Registry Data Structures

3.1 Data structure overview

This chapter describes the semantics of the data structures that are specified by the UDDI API Schema. Refinements that are specific to individual APIs are described in Chapter 5 *UDDI Programmers API's*.

As described in Section 1.6 *Representing Information within UDDI*, the information that makes up a UDDI registry consists of instances of four core data structure types, the businessEntity, the businessService, the bindingTemplate and the tModel, together with instances of additional data structure types defined in the UDDI API Schema.

The four core types and their relationships are shown in a simplified diagram in Figure 1 and are explained in detail in this chapter.

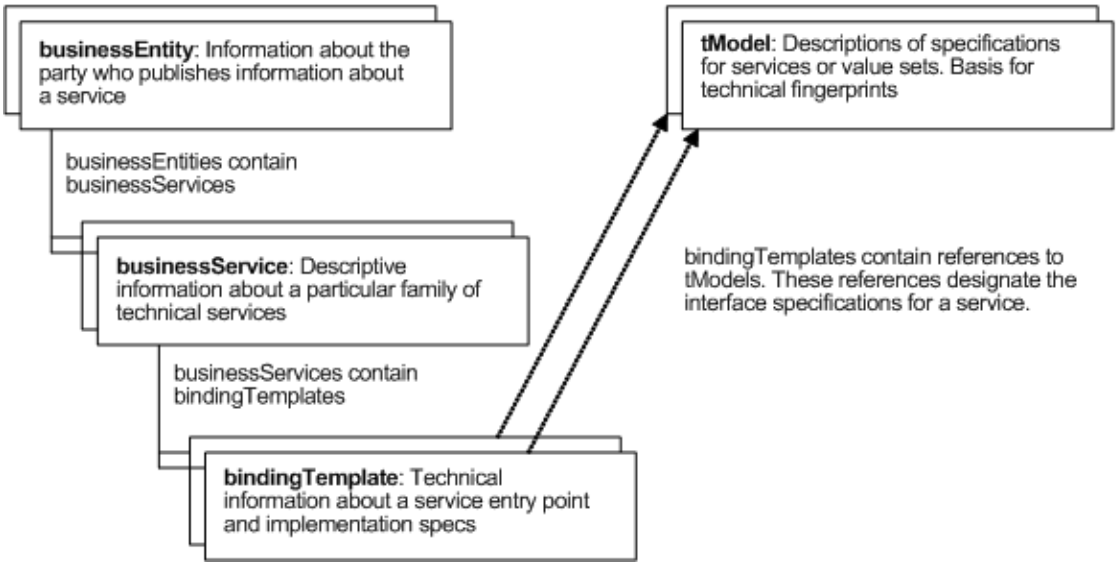


Figure 1 - UDDI core data structures

The schema also defines a number of request and response structures, each of which contain the core structures, references to the core structures, or summary versions of them; see Chapter 5 *UDDI Programmers API's* for details.

3.2 Design Principles

Each of the core data structure types is used to express specific types of data, arranged in the relationship shown in Figure 1. A particular instance of an individual fact or set of related facts is expressed using XML according to the definition of these core types. For instance, two separate businesses may publish information in a UDDI registry about Web services they offer. Information describing each business and its Web services all exists as separate instances of the core data structures stored within the UDDI registry.

3.2.1 Keys as unique identifiers

Instances of many data structures in UDDI, including all of the core data structures are kept separately, and are accessed individually by way of unique identifiers called keys. An instance in the registry gets its keys at the time it is first published. Publishers may assign keys; if they don't, the UDDI node MUST assign them. See Section 4.4 *About uddiKeys*.

3.2.2 Containment and references

The core data structures are sensitive to the containment relationships found in the UDDI API schema and shown in Figure 1. The businessEntity structure contains one or more distinct businessService structures. Similarly, individual businessService structures contain specific instances of bindingTemplate structures.

It is important to note that no single instance of an entity is ever "contained" by more than one containing entity. This means, for example, that only one specific businessEntity structure (identified by its unique key value) will ever contain a specific instance of a businessService structure (also identified by its own unique key).

References, on the other hand, operate differently. We can see an example of this in Figure 1 where the bindingTemplate entities refer to instances of tModel entities. References to a given entity can occur multiple times, as needed.

Determining what is a reference and what is the key for a specific entity is straightforward. Each kind of keyed entity has an attribute whose type is a corresponding type of key. For example, businessEntity has a businessKey attribute and a businessService has a serviceKey attribute. The value of this attribute is the entity's key. All other keys are references or containment relationships. Taking the bindingTemplate as an example, the tModelKey that occurs in its inner structure is a reference and the serviceKey that occurs in the bindingTemplate is a containment relationship.

3.2.3 Collections

Many elements in the UDDI API Schema may occur multiple times. Those elements that do not have a complex inner structure, for example, name and description, are provided in a list. Elements that do have a more complex inner structure are usually grouped in their own container element. For example, the contacts structure is a container where one or more contact structures reside.

3.2.4 Optional attributes

In the data structure elements of the UDDI API Schema, there are many optional attributes, for example, keyName and useType. Most optional attributes have defaults of empty string (""). During schema assessment, this produces a single representation for an omitted or empty string in an optional attribute. Consider the following two keyedReferences:

```
<keyedReference
    tModelKey="uddi:uddi.org:ubr:categorization:iso3166"
    keyName=""
    keyValue="US-CA" />
<keyedReference
    tModelKey="uddi:uddi.org:ubr:categorization:iso3166"
    keyValue="US-CA" />
```

Semantically speaking from the perspective of UDDI, omitted attributes are identical to empty attributes. However, with respect to signing, specifically, canonicalization, omitted attributes are different from empty attributes. Therefore, the digital signatures of the above two keyedReferences are different, even though clients would consider the two keyedReferences be identical.

The difference, from a perspective of canonicalization, puts additional burden on clients in publishing entities. As a result, when applicable, the data structure elements of UDDI API Schema define default values for optional attributes, so that omitted attributes are treated as attributes with default value with respect to signing.

The exceptions are xml:lang and keyValue in addressLine. Both prohibit empty string. Hence, the ambiguity discussed above is not applicable. In the case of xml:lang, empty string is not a valid language code. In the case of keyValue in addressLine, the definition of keyValue requires the string to have a minimal length of one.

3.3 businessEntity Structure

Each businessEntity entity contains descriptive information about a business or organization and, through its contained businessService entities, information about the services that it offers. From an XML standpoint, the businessEntity is the top-level data structure that holds descriptive information about the business or organization it describes. Each contained businessService describes a logical service offered by the business or organization. Similarly, each bindingTemplate contained within a given businessService provides the technical description of a Web service that belongs to the logical service that is described by the businessService.

3.3.1 Structure diagram

Attributes

Name	Use
businessKey	optional

3.3.2 Documentation

A given instance of the businessEntity structure is uniquely identified by its **businessKey**. When a businessEntity is published within a UDDI registry, the businessKey MUST be omitted

if the publisher wants the registry to generate a key. When a businessEntity is retrieved from a UDDI registry, the businessKey MUST be present.

discoveryURLs is a list of Uniform Resource Locators (URL) that point to alternate, file based service discovery mechanisms.

Simple textual information about the businessEntity, potentially in multiple languages, is given by its **name**, short business **description** and **contacts**. The required, non-empty name and the optional description can occur multiple times. **contacts** is a simple list of single contact information.

businessServices is a list of business services provided by the businessEntity.

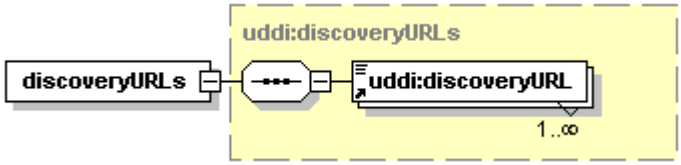
In addition to the businessKey, that uniquely identifies the businessEntity within the registry, the **identifierBag** contains a list of other identifiers, each valid in its own identifier system. Examples of identifiers are a tax identifier or D-U-N-S® number.

The **categoryBag** contains a list of business categories that each describes a specific business aspect of the businessEntity. Examples of categories are industry, product category or geographic region.

A businessEntity entity MAY be digitally signed using XML digital signatures. When a businessEntity is signed, each digital signature MUST be provided by its own **dsig:Signature** element. Appendix I *Support for XML Digital Signatures* covers the use of this element in accordance with the XML-Signature specification.

3.3.2.1 discoveryURLs

The discoveryURLs structure is a simple container of one or more **discoveryURL** elements.



3.3.2.2 discoveryURL

A discoveryURL is a URL that points to Web addressable (via HTTP GET) discovery documents. The expected return document is not defined. Rather, a framework for establishing conventions is provided, and a particular convention is defined within this specification.

Attributes

Name	Use
useType	optional

Each individual discoveryURL MAY be adorned with a **useType** attribute that designates the name of the convention that the referenced document follows. A reserved convention value is "businessEntity". It is RECOMMENDED that discoveryURLs qualified with this value point to XML documents of the type businessEntity, as defined in the UDDI API Schema.

An example of a discoveryURL, generated by a UDDI node that is accessible at www.example.com and rendered by the publisher of the businessEntity that is identified by the businessKey "uddi:example.com:registry:sales:53", is:

```
<discoveryURL useType="businessEntity">
    http://www.example.com?businessKey=uddi:example.com:registry:sales:53
</discoveryURL>
```

Another reserved value for discoveryURL is "homepage". Adorning a discoveryURL with this value signifies that a business's homepage can be discovered at that URL.

3.3.2.3 name

A businessEntity MAY contain more than one name. Multiple names are useful, for example, in order to specify both the legal name and a known abbreviation of a businessEntity, or in order to support romanization (see Appendix D *Internationalization*).

Attributes

Name	Use
xml:lang	optional

When a name is expressed in a specific language (such as the language into which a name has been romanized), it SHOULD carry the **xml:lang** attribute to signify this. When a name does not have an associated language (such as a neologism not associated with a particular language), the xml:lang attribute SHOULD be omitted.

As is defined in the XML specification, an occurrence of the xml:lang attribute indicates that the content to which it applies (namely the element on which it is found and to all its children, unless subsequently overridden) is to be interpreted as being in a certain natural language. Legal values for such attributes conform to RFC 3066[5] with one exception: UDDI imposes a maximum length of 26 characters.

As is the case for RFC 3066, all tags are to be treated as case insensitive; there exist conventions for capitalization of some of them, but these should not be taken to carry meaning. For instance, [ISO 3166] recommends that country codes are capitalized (MN Mongolia), while [ISO 639] recommends that language codes are written in lower case (mn Mongolian).

Examples include: "EN-us", "FR-ca".

3.3.2.4 description

A businessEntity can contain several descriptions, for example, in different languages.

Attributes

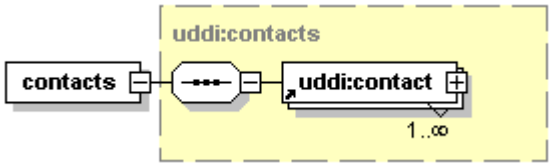
Name	Use
xml:lang	optional

In order to signify the language in which the descriptions are expressed, they MAY carry **xml:lang** values. There is no restriction on the number of descriptions or on what xml:lang value that they may have.

3.3.2.5 contacts

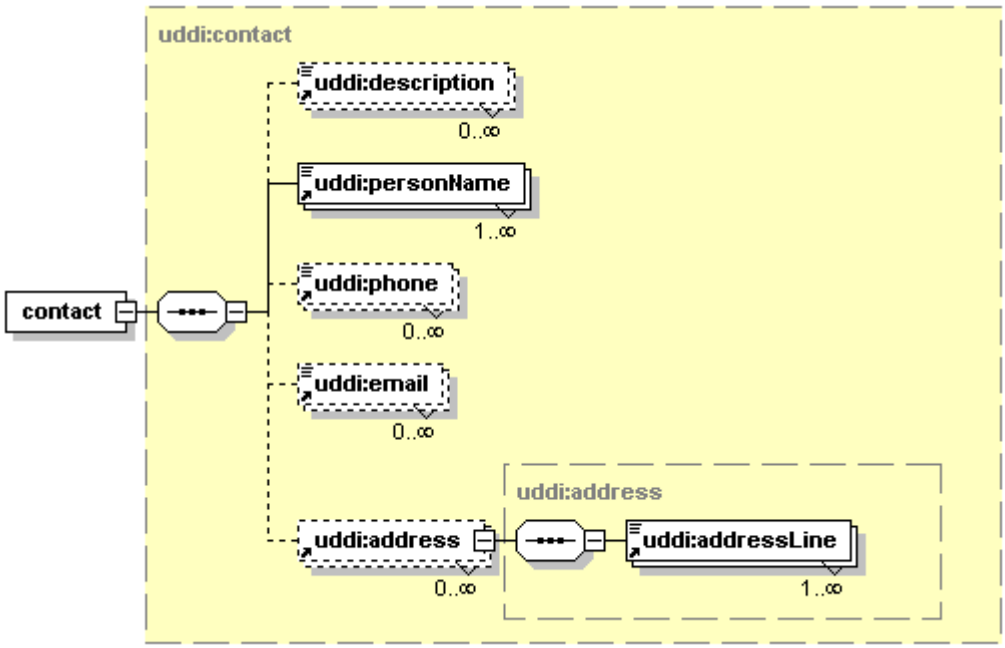
The contacts structure itself is a simple collection of one or more **contact** structures.

[5] Note that the language type in the XML Schema Recommendation does not conform to RFC 3066. It is necessary to use schema assessment conforming to the errata for XML Schema (see http://www.w3.org/2001/05/xmlschema-errata#e2-25)



3.3.2.6 contact

The contact structure records contact information for a person or a job role within the businessEntity so that someone who finds the information can make human contact for any purpose. This information consists of one or more optional elements, along with a person's name. Contact information exists by containment relationship alone; the contact structure does not provide keys for tracking individual contact instances.



Attributes

Name	Use
useType	optional

The **useType** attribute is used to describe the type of contact in unstructured text. Suggested examples include "technical questions", "technical contact", "establish account", "sales contact", etc.

description is used to describe how the contact information should be used. Publishing several descriptions, e.g. in different languages, is supported. To signify the language in which the descriptions are expressed, they MAY carry **xml:lang** values.

personName is the name of the person or name of the job role supporting the contact. Publishing several names, e.g. for romanization purposes, is supported.

Attributes

Name	Use
xml:lang	optional

In order to signify the contextual language (if any) in which a given name is expressed in (such as the language into which a name has been romanized), it SHOULD carry the **xml:lang** attribute See Section 3.3.2.3 *name* for details on using xml:lang values in name elements. There is no restriction on the number of personNames or what xml:lang value each may have. An example for a role might be:

```
<contact useType="Technical support">
    <personName>Administrator</personName>
    …
</contact>
```

phone is used to hold telephone numbers for the contact. This element MAY be adorned with an optional **useType** attribute for descriptive purposes.

email is the email address for the contact. This element MAY be adorned with an optional **useType** attribute for descriptive purposes.

address is the postal address for the contact.

3.3.2.7 address

address represents the contact's postal address, in a form suitable for addressing an envelope. The address structure is a simple list of address lines.

Attributes

Name	Use
xml:lang	optional
useType	optional
sortCode	optional
tModelKey	optional

Address structures have four optional attributes.

The **xml:lang** value describes the language the address is expressed in. There is no restriction on the number of addresses or what xml:lang value they may have. Publication of addresses in several languages, e.g. for use in multilingual countries, is supported. See Appendix D *Internationalization* for an example.

The **useType** describes the address' type in unstructured text. Suggested examples include "headquarters", "sales office", "billing department", etc.

The **sortCode** attribute is deprecated because of the guarantee of preserving the document order (see Section 4.5.3 *Preservation of Document Order*). In order to achieve this behavior, the data has just to be published in the desired order.

The **tModelKey** is a tModel reference that specifies that keyName keyValue pairs given by subsequent addressLine elements, if addressLine elements are present at all, are to be interpreted by the address structure associated with the tModel that is referenced. For a description of how to use tModels in order to give the addressLine list structure and meaning, see Appendix D *Internationalization*.

3.3.2.8 addressLine

addressLine contains a part of the actual address in text form.

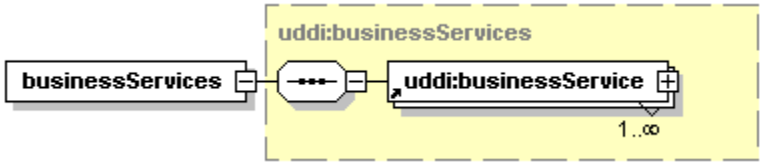
Attributes

Name	Use
keyName	optional
keyValue	optional

Each addressLine element MAY be adorned with two optional descriptive attributes, **keyName** and **keyValue**. Both attributes MUST be present in each address line if a tModelKey is specified in the address structure. When no tModelKey is provided for the address structure, the keyName and keyValue attributes have no defined meaning.

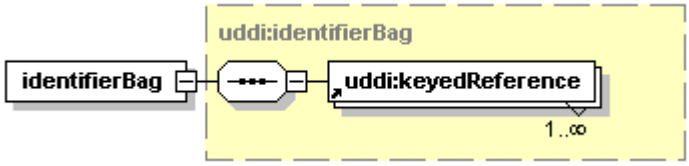
3.3.2.9 businessServices

The businessServices structure is used to describe families of Web services. This simple container holds one or more businessService entities (see Section 3.4 *businessService structure*).



3.3.2.10 identifierBag

The optional identifierBag element allows businessEntity structures to be identified according to published identifier systems, for example, Dun & Bradstreet D-U-N-S® numbers or tax identifiers.



An identifierBag is a list of one or more **keyedReference** structures, each representing a single identification.

For a full description on how to establish an identity, see Appendix E *Using Identifiers*.

3.3.2.11 keyedReference (in identifierBags)

A keyedReference, when included in an identifierBag, represents an identifier of a specific identifier system.

Attributes

Name	Use

tModelKey	required
keyName	optional
keyValue	required

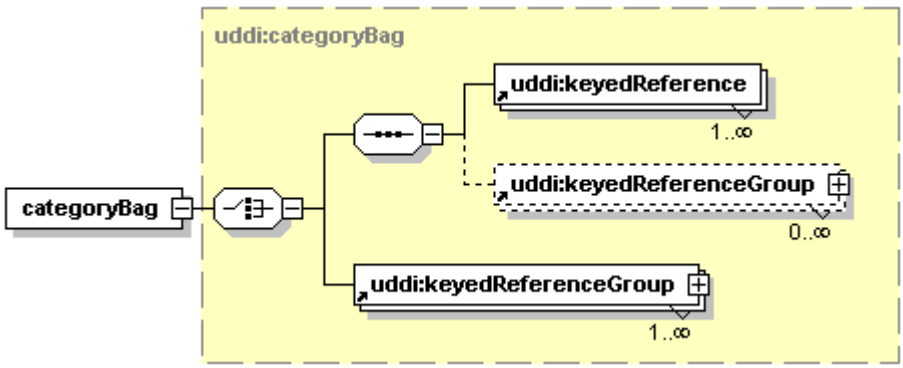
The keyedReference consists of the three attributes **tModelKey**, **keyName** and **keyValue**. The required tModelKey refers to the tModel that represents the identifier system, and the required keyValue contains the actual identifier within this system. The optional keyName MAY be used to provide a descriptive name for the identifier. Omitted keyNames are treated as empty keyNames.

For example, identifying SAP AG by its Dun & Bradstreet D-U-N-S® Number, using the corresponding tModelKey within the UDDI Business Registry, is done as follows:

```
<identifierBag>
    <keyedReference
        tModelKey="uddi:uddi.org:ubr:identifier:dnb.com:d-u-n-s"
        keyName="SAP AG"
        keyValue="31-626-8655" />
</identifierBag>
```

3.3.2.12 categoryBag

The optional categoryBag element allows businessEntity structures to be categorized according to published categorization systems. For example, a businessEntity might contain UNSPSC product and service categorizations that describe its product and service offering and ISO 3166 geographical regions that describe the geographical area where these products and services are offered.



Similar to the identifierBag, a categoryBag contains a simple list of **keyedReference** structures, each containing a single categorization. The categoryBag MAY also contain a simple list of **keyedReferenceGroup** structures. At least one keyedReference or one keyedReferenceGroup MUST be provided within the categoryBag.

For a full description of how to establish a categorization, see Appendix F *Using Categorization*.

3.3.2.13 keyedReference (in categoryBags)

As within an identifierBag (see Section 3.3.2.13 *keyedReference (in identifierBags)*), a keyedReference contains the three attributes **tModelKey**, **keyName** and **keyValue**. The required tModelKey refers to the tModel that represents the categorization system, and the required keyValue contains the actual categorization within this system. The optional keyName can be used to provide a descriptive name of the categorization. Omitted keyNames are treated as empty keyNames. A keyName MUST be provided in a keyedReference if its

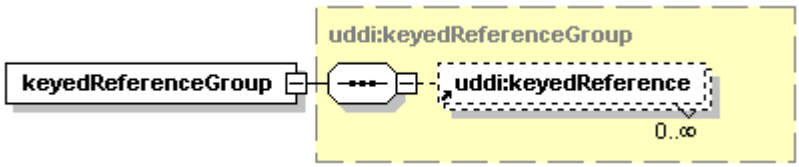
tModelKey refers to the general_keywords category system (see also Section 5.1.7 *Matching Rules for keyedReferences and keyedReferenceGroups*).

For example, in order to categorize a businessEntity as offering goods and services in California, USA, using the corresponding ISO 3166 tModelKey within the UDDI Business Registry, one would add the following keyedReference to the businessEntity's categoryBag:

```
<keyedReference
    tModelKey="uddi:uddi.org:ubr:categorization:iso3166"
    keyName="California, USA"
    keyValue="US-CA" />
```

3.3.2.14 keyedReferenceGroup

A keyedReferenceGroup, by itself, is a simple list of keyedReference structures that logically belong together.



Attributes

Name	Use
tModelKey	required

The keyedReferenceGroup MUST contain a **tModelKey** attribute that specifies the structure and meaning of the keyedReferences contained in the keyedReferenceGroup. A keyedReferenceGroup MUST also contain at least one keyedReference when published.

For example, to categorize a businessEntity as being located at the geodetic point that is specified by the latitude/longitude pair 49.6827/8.2952 using the corresponding World Geodetic System 1984 (WGS 84) tModelKey within the UDDI Business Registry, one would add the following keyedReferenceGroup to the businessEntity's categoryBag:

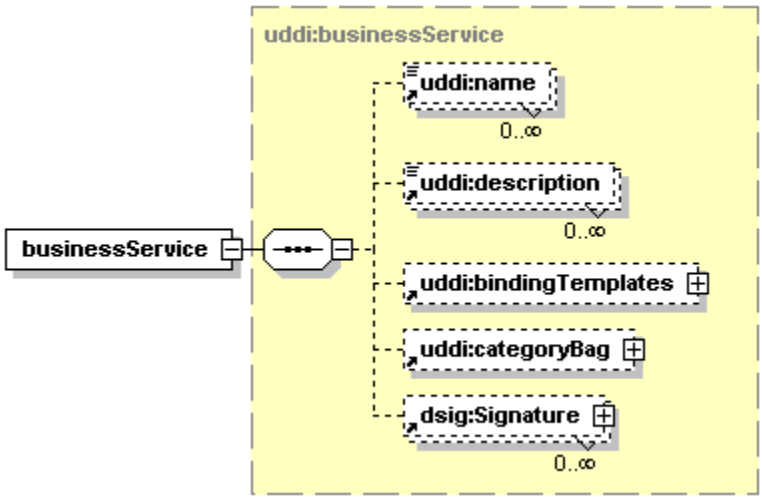
```
<keyedReferenceGroup tModelKey="uddi:uddi.org:ubr:categorizationGroup:wgs84" >
    <keyedReference
        tModelKey="uddi:uddi.org:ubr:categorization:wgs84:latitude"
        keyName="WGS 84 Latitude"
        keyValue="+49.682700" />
     <keyedReference
         tModelKey="uddi:uddi.org:ubr:categorization:wgs84:longitude"
         keyName="WGS 84 Longitude"
         keyValue="+008.295200" />
</keyedReferenceGroup>
```

3.4 businessService Structure

The businessService structure represents a logical service and contains descriptive information in business terms. A businessService is the logical child of a single businessEntity, the provider of this businessService. Technical information about the businessService is found in the contained bindingTemplate entities.

In some cases, businesses would like to share or reuse services, e.g. when a large enterprise publishes separate businessEntity structures. This can be done by using the businessService structure as a *projection* to a published businessService, as explained below.

3.4.1 Structure Diagram



Attributes

Name	Use
serviceKey	optional
businessKey	optional

3.4.2 Documentation

A given businessService entity is uniquely identified by its **serviceKey**. When a businessService is published within a UDDI registry, the serviceKey MUST be omitted if the publisher wants the registry to generate a key. When a businessService is retrieved from a UDDI registry, the serviceKey MUST be present.

The **businessKey** attribute uniquely identifies the businessEntity which is the provider of the businessService. Every businessService is "contained" in exactly one businessEntity.

When a businessService is published within a UDDI registry, the businessKey MAY be omitted if the businessService is a part of a fully expressed businessEntity element. When a businessService is retrieved from a UDDI registry, the businessKey MUST be present. This behavior provides the ability to browse through the containment relationships given any of the core elements as a starting point.

The businessKey may differ from the publishing businessEntity's businessKey. This indicates a service projection. A service projection allows a business or organization to include in its businessEntity a businessService offered by some other business or organization. A projected businessService is made a part of a businessEntity by reference as opposed to by containment. Projections to the same service can be made in any number of business entities.

Simple textual information about the businessService, potentially in multiple languages, is given by its **name** and short service **description**. The non-empty name, required except when indicating a service projection, and the optional description can occur multiple times. More information about the structure of the name and description elements can be found in Section 3.3 *businessEntity Structure*.

bindingTemplates is a list of technical descriptions for the Web services provided.

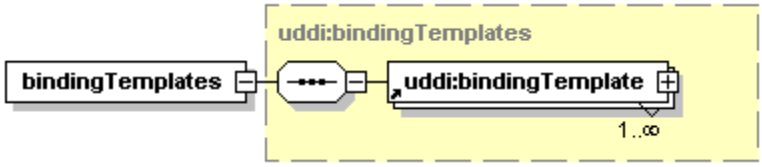
The **categoryBag** contains a list of business categories that each describes a specific business aspect of the businessService (e.g. industry, product category or geographic region)

and is valid in its own category system. More information about the categoryBag element can be found in Section 3.3 *businessEntity Structure*.

A businessService entity MAY be digitally signed using XML digital signatures. When a businessService is signed, each digital signature MUST be provided by its own **dsig:Signature** element. Appendix I *Support for XML Digital Signature* covers the use of this element in accordance with the XML-Signature specification.

3.4.2.1 bindingTemplates

The **bindingTemplates** structure holds, for a given businessService, the bindingTemplate entities that provide the technical descriptions of the Web services that constitute the businessService.



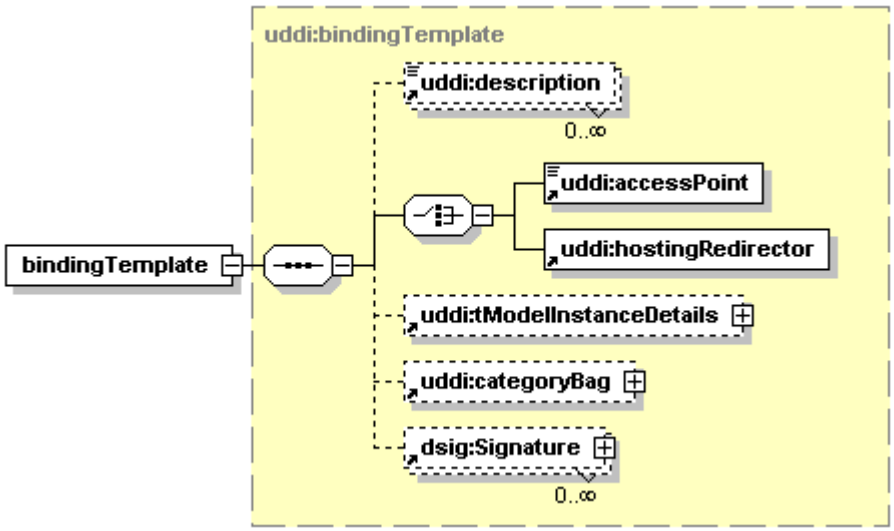
See Section 3.5 *bindingTemplate structure* for details on bindingTemplates.

3.5 bindingTemplate Structure

Technical descriptions of Web services are provided by bindingTemplate entities. Each bindingTemplate describes an instance of a Web service offered at a particular network address, typically given in the form of a URL. The bindingTemplate also describes the type of Web service being offered using references to tModels, application-specific parameters, and settings.

Each bindingTemplate is contained in a businessService.

3.5.1 Structure Diagram



Attributes

Name	Use
bindingKey	optional
serviceKey	optional

3.5.2 Documentation

A given bindingTemplate entity is uniquely identified by its **bindingKey**. When a bindingTemplate is published within a UDDI registry, the bindingKey MUST be omitted if the publisher wants the registry to generate a key. When a bindingTemplate is retrieved from a UDDI registry, the bindingKey MUST be present.

The **serviceKey** attribute uniquely identifies the businessService that contains the bindingTemplate. When a bindingTemplate is published within a UDDI registry, the serviceKey MAY be omitted if the bindingTemplate is a part of a fully expressed businessService element. When a bindingTemplate is retrieved from a UDDI registry, the serviceKey MUST be present.

Simple textual information about the bindingTemplate, potentially in multiple languages, is given by its short binding **description**. It is optional and can occur multiple times. More information about the structure of the description element can be found in Section 3.3 *businessEntity structure*.

The **accessPoint** is a string used to convey the network address suitable for invoking the Web service being described. This is typically a URL but may be an electronic mail address, or even a telephone number. No assumptions about the type of data in this field can be made without first understanding the technical requirements associated with the Web service.

The **hostingRedirector** is a deprecated element, since its functionality is now covered by the accessPoint. For backward-compatibility, it can still be used, but it is not recommended. See the set of UDDI Version 2 Specifications for a description on hostingRedirector.

Either an accessPoint or a hostingRedirector must be provided within a bindingTemplate.

The **tModelInstanceDetails** structure is a list of one or more tModelInstanceInfo elements. The collection of tModelKey attributes found in the tModelInstanceInfo elements together form the "technical fingerprint" of a Web service that can be used to identify compatible services.

The **categoryBag** contains a list of categorizations that each describes a specific aspect of the bindingTemplate and is valid in its own category system. A categoryBag in a bindingTemplate can be used, for example, to indicate that the Web service described by the bindingTemplate has the status "test" or "production". More information about the structure of the categoryBag element can be found in Section 3.3 *businessEntity Structure*.

A bindingTemplate entity MAY be digitally signed using XML digital signatures. When a bindingTemplate is signed, each digital signature MUST be provided by its own **dsig:Signature** element. Appendix I *Support for XML Digital Signature* covers the use of this element in accordance with the XML-Signature specification.

3.5.2.1 accessPoint

The accessPoint element is an attribute-qualified URI, typically a URL, representing the network address of the Web service being described. The notion of Web service seen here is fairly abstract and many types of network addresses are accommodated.

Attributes

Name	Use
useType	optional

The purpose of the optional attribute **useType** is to facilitate the description of several types of accessPoints.

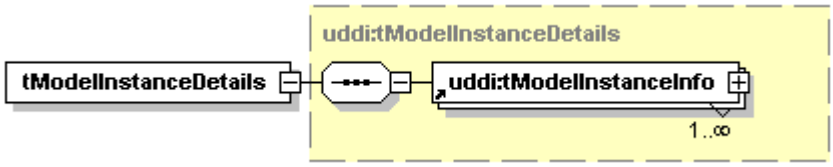
The following useType attributes values are pre-defined by UDDI:

- **endPoint**: designates that the accessPoint points to the actual service endpoint, i.e. the network address at which the Web service can be invoked,
- **bindingTemplate**: designates that the accessPoint contains a bindingKey that points to a different bindingTemplate entry. The value in providing this facility is seen when a business or entity wants to expose a service description (e.g. advertise that they have a service available that suits a specific purpose) that is actually a service that is described in a separate bindingTemplate record. This might occur when many service descriptions could benefit from a single service description,
- **hostingRedirector**: designates that the accessPoint can only be determined by querying another UDDI registry. This might occur when a service is remotely hosted.
- **wsdlDeployment**: designates that the accessPoint points to a remotely hosted WSDL document that already contains the necessary binding information, including the actual service endpoint.

The useType attribute may contain other values than the four listed above. See Appendix B *Using and Extending the useType Attribute* for more information.

3.5.2.2 tModelInstanceDetails

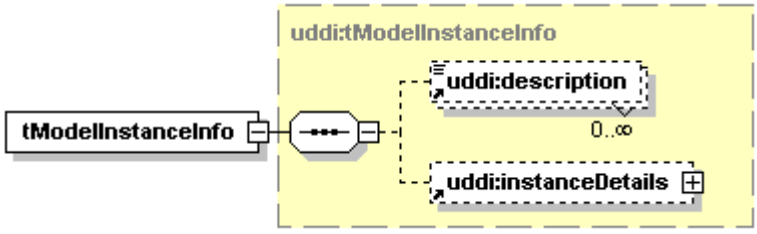
This structure is a container for one or more **tModelInstanceInfo** structures.



When a bindingTemplate is published it SHOULD contain, a tModelInstanceDetails element that in turn contains in its tModelInstanceInfo structures one or more tModel references. This arbitrarily ordered collection of references is called the "technical fingerprint" of the Web service. It indicates that the Web service being described complies with the specific and identifiable specifications implied by the tModelKey values provided. During an inquiry, interested parties can use this information to look for bindingTemplate entities that contain a specific fingerprint or partial fingerprint.

3.5.2.3 tModelInstanceInfo

Each tModelInstanceInfo structure represents bindingTemplate entity-specific details for each tModel referenced.



Attributes

Name	Use
tModelKey	required

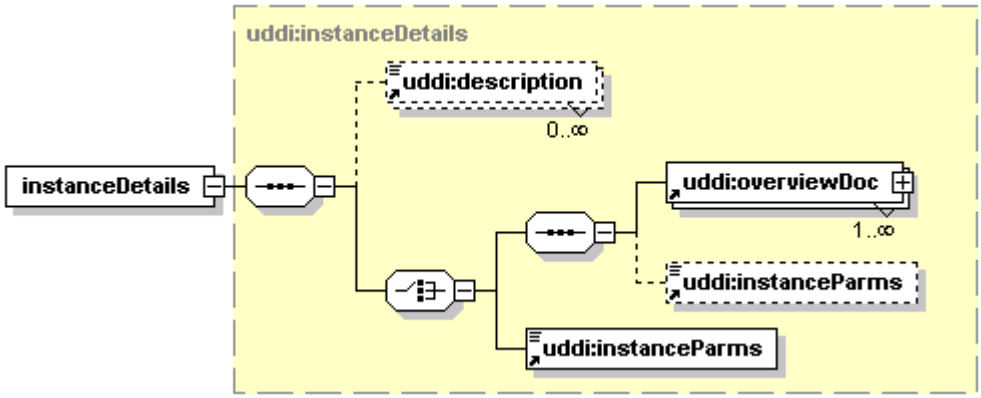
The required **tModelKey** attribute references a tModel that represents a specification with which the Web service represented by the containing bindingTemplate complies.

The **description** is an optional repeating element. Each description, optionally qualified by an xml:lang attribute, describes what role the tModel plays in the overall service description.

The optional **instanceDetails** element can be used when tModel-specific settings or other descriptive information are required either to describe a tModel specific component of a service description or to support services that require additional technical data support (e.g. via settings or other handshake operations).

3.5.2.4 instanceDetails

This structure holds service instance-specific information that is required to either understand the service implementation details relative to a specific tModel, or to provide further parameter and settings support.



The **description** is an optional repeating element. Each description, optionally qualified by an xml:lang attribute, describes the purpose and/or use of the particular instanceDetails entry.

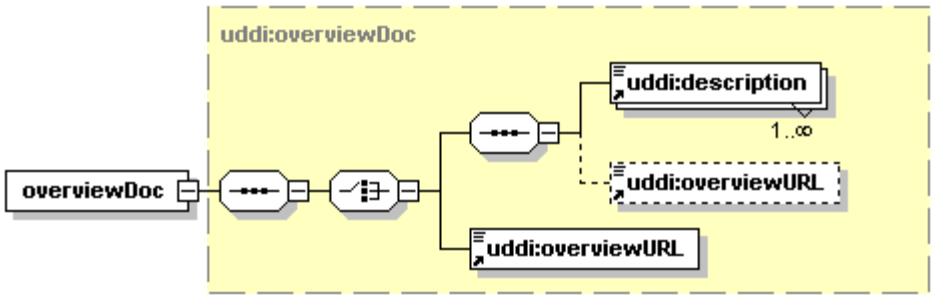
The **overviewDoc** is a mandatory repeating element, used to house references to remote descriptive information or instructions related to the use of a particular tModel and its instanceParms. Multiple overviewDoc elements are useful, for example, to handle alternative representations of the documentation.

The **instanceParms** is an optional element of type string, used to locally contain settings or parameters related to the proper use of a tModelInstanceInfo. The suggested format is a namespace-qualified XML document so that the settings or parameters can be found in the XML documents elements and attributes.

At least one overviewDoc or instanceParms MUST be provided within the instanceDetails.

3.5.2.5 overviewDoc

This structure describes overview information about a particular tModel use within a bindingTemplate.



The **description** is a mandatory repeating element. Each description, optionally qualified by an xml:lang attribute, holds a short descriptive overview of how a particular tModel is to be used.

The optional **overviewURL** is to be used to hold a URL that refers to a long form of an overview document that covers the way a particular tModel is used as a component of an overall Web service description.

At least one description or an overviewURL MUST be provided within the overviewDoc.

3.5.2.6 overviewURL

The RECOMMENDED format for the overviewURL is a URI that is suitable for retrieving the actual overview document with an HTTP GET operation, for example, via a Web browser.

Attributes

Name	Use
useType	optional

The optional **useType** attribute is used to provide information about the type of document at that URL. One common value used in the useType attribute is "text". Using this value denotes that the overviewURL contains additional textual information. The content of the useType attribute may contain other values. See Appendix B *Using and Extending the useType Attribute* for more information.

3.6 tModel Structure

Making it easy to describe Web services in ways that are meaningful enough to be useful during searches is an important goal of UDDI. Another goal is to provide a facility to make these descriptions complete enough that people and programs can discover how to interact with Web services they do not know much about. To do this, there needs to be a way to mark a description with information that designates how it behaves, what conventions it follows, and what specifications or standards the service complies with.

Providing the ability to describe compliance with specifications, concepts, or even shared design is one of the roles that the tModel structure fills.

Each tModel instance is a keyed entity in UDDI. In a general sense, the purpose of tModel entities is to provide a reference system based on abstraction. There are two primary uses for tModel entities: as sources for determining compatibility of Web services and as keyed namespace references.

3.6.1 Common tModel uses

There are several places within a businessEntity that can refer to tModels. References to the same tModel instance can be found in many businessEntity structures. tModel references also occur in various API calls.

Section 3.6 *tModel Structure* gives an overview of the different types of tModels.

3.6.1.1 Defining the technical fingerprint

One common use for tModel entities is to represent technical specifications or concepts. For example, a tModel can be used to represent a specification that defines wire protocols, interchange formats and interchange sequencing rules. Examples can be seen in the RosettaNet Partner Interface Processes[6] specification, the Open Applications Group Integration Specification[7] and various Electronic Document Interchange (EDI) efforts.

[6] See http://www.rosettanet.org

[7] See http://www.openapplications.org

Software that communicates with other software invariably adheres to some pre-agreed specifications. In situations where this is true, the designers of the specifications can establish a unique technical identity within a UDDI registry by publishing information about the specification in a tModel. While the main reason of registering a tModel with a specific UDDI registry is to define its identity, the actual specification or set of documents that describes the concept of a tModel is not a part of the registry and is remotely referenced using the overviewDoc structure. Publishers SHOULD choose well-known formats and description languages for the documents that describe the concept each tModel represents.

Once a tModel is published, other parties can express the availability of Web services that are compliant with a specification the tModel represents by simply including a reference to the tModel – i.e., its tModelKey – in their technical service descriptions bindingTemplate data.

This approach facilitates searching for registered Web services that are compatible with a particular specification. Once the proper tModelKey value is known, it is easy to discover that a particular businessEntity has registered a Web service that references the tModel. In this way, the tModelKey becomes a technical fingerprint that is unique to a given specification.

3.6.1.2 Defining value sets

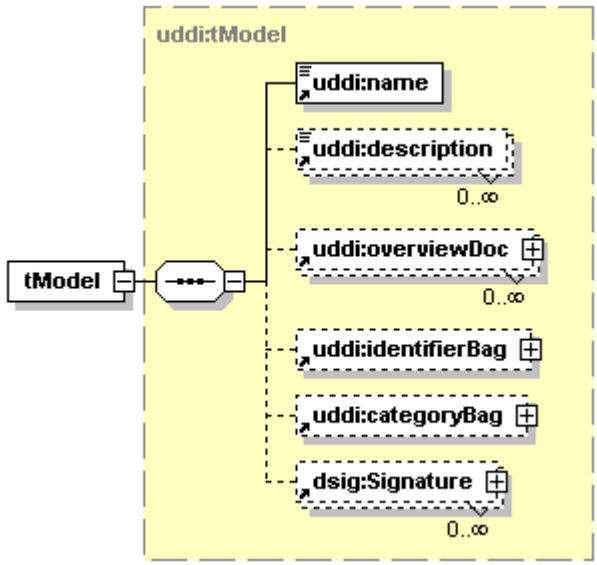
The second general use for tModel entities is within the identifierBag, categoryBag, address and publisherAssertion structures that are used to specify organizational identity and various categories. Used in this context, a tModel represents the system of values used to identify or categorize UDDI entities.

For example, to represent the fact that a business described by a businessEntity has a particular US Tax identifier, a keyedReference is placed into the identifierBag of the businessEntity. The keyedReference has a keyValue that is the tax ID and refers to the tModel that means "the system of US Tax code identifiers". Together, the keyValue and the tModel reference specify a particular value in a particular system of values.

3.6.1.3 Defining a find qualifier

The third general use for tModel entities is to represent find qualifiers. Find qualifiers are values that modify how the find_xx APIs work. For example, to cause find_business to sort its results in the order in which they were published, the uddi:uddi.org:findqualifier:sortbydateasc may be specified. See Section 5.1.4 *Find Qualifiers* for details.

3.6.2 Structure diagram



Attributes

Name	Use
tModelKey	optional
deleted	optional

3.6.3 Documentation

A given tModel entity is uniquely identified by its **tModelKey**. When a tModel is published within a UDDI registry, the tModelKey MUST be omitted if the publisher wants the registry to generate a key. When a tModel is retrieved from a UDDI registry, the tModelKey MUST be present.

In retrieved tModel data, the **deleted** attribute, an information-only field, indicates whether the tModel is logically deleted. The two allowed values for this attribute are "true" and "false".

Simple textual information about the tModel, potentially in multiple languages, is given by its **name** and short **description**. While the tModel has exactly one non-empty name, it can have zero or more descriptions. **The name SHOULD be formatted as a URI and, as a consequence, the xml:lang attribute of the name element SHOULD NOT be used.** More information about the structure of the name and description elements can be found in Section 3.3 *businessEntity structure*.

The **overviewDoc** is an optional repeating element, used to house references to remote descriptive information or instructions related to the tModel. For more information about the structure of the overviewDoc v, see Section 3.5 *bindingTemplate Structure*.

The optional **useType** attribute contained in the **overviewURL** of the **overviewDoc** is used to provide information about the type of document at that URL. One common value used in the useType attribute is "text". Using this value denotes that the overviewURL contains additional textual information. Another common value is "wsdlInterface", which is used to designate that

the overviewURL contains a WSDL interface document that can be re-used by many implementations. The content of the useType attribute may contain other values. See Appendix B *Using and Extending the useType Attribute* for more information.

In addition to the tModelKey that uniquely identifies the tModel within the registry, the **identifierBag** contains a list of logical identifiers, each valid in its own identifier system. For more information about identifierBags, see Section 3.3 *businessEntity Structure*.

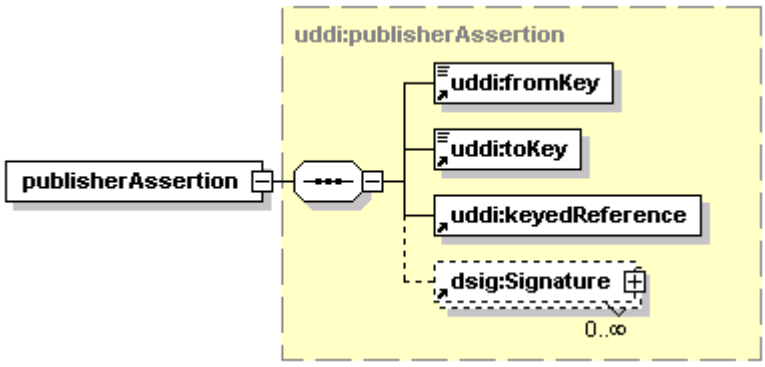
The **categoryBag** contains a list of categories that describe specific aspects of the tModel (e.g. its technical type). Each category is valid in its own category system. For more information about categoryBags, see Section 3.3 *businessEntity structure*.

A tModel entity MAY be digitally signed using XML digital signatures. When a tModel is signed, each digital signature MUST be provided by its own **dsig:Signature** element. Appendix I *Support for XML Digital Signatures* covers the use of this element in accordance with the XML-Signature specification.

3.7 publisherAssertion Structure

Many businesses and organizations are not effectively represented by a single businessEntity, because their description and discovery are likely to be diverse. Examples include corporations with a variety of subsidiaries, private exchanges with sets of suppliers and their customers and industry consortiums with their members. An obvious solution is to publish several businessEntity structures. Such a set of businessEntity structures represents a more or less coupled community whose members often would like to make some of their relationships visible in their UDDI registrations. This may be accomplished by using the publisherAssertion structure. To eliminate the possibility that one publisher claims a relationship to another that is not reciprocated, both publishers must publish identical assertions for the relationship to become visible. More detailed information about relationships and publisher assertions is given in Appendix A *Relationships and Publisher Assertions*.

3.7.1 Structure Diagram



3.7.2 Documentation

The two businessEntity instances between which an assertion is made are uniquely identified by the required **fromKey** and **toKey** elements. The **keyedReference** describes the relationship between the businessEntity elements identified by fromKey and toKey. Similar to the general behavior of a keyedReference in a categoryBag (see full description in Section 3.3 *businessEntity Structure*), the included tModelKey uniquely identifies the relationship type system and the keyName keyValue pair designate a specific relationship type within this value set. Omitted keyNames are treated as empty keyNames.

A publisherAssertion entity MAY be digitally signed using XML digital signatures. When a publisherAssertion is signed, each digital signature MUST be provided by its own

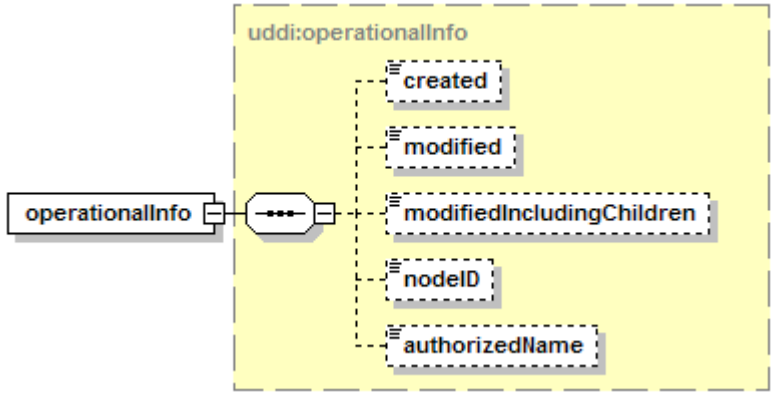
dsig:Signature element. Appendix I *Support For XML Digital Signatures* covers the use of this element in accordance with the XML-Signature specification.

3.8 operationalInfo Structure

Information about a publishing operation is captured whenever a UDDI core data structure is published. This data includes the date and time that the data structure was created and modified, the identifier of the UDDI node at which the publish operation took place, and the identity of the publisher. Operational information for a UDDI data structure is made accessible using the get_operationalInfo inquiry API. See Section 5.1.16 *get_operationalInfo*.

The operationalInfo structure is used to convey the operational information for the UDDI core data structures, that is, the businessEntity, businessService, bindingTemplate and tModel structures.

3.8.1 Structure diagram



Attributes

Name	Use
entityKey	required

3.8.2 Documentation

The UDDI entity with which the operationalInfo is associated is uniquely identified by the required **entityKey** attribute.

The **created** element indicates the instant in time at which the entity with which the operationalInfo is associated first appeared in a registry.

The **modified** element indicates the instant in time at which the entity with which the operationalInfo is associated was last changed by a save operation that may have updated its content. This will initially be equivalent to the value of the created element, but will diverge as changes are made over time.

Some UDDI core data structures are containers of other UDDI core data structures. For instance, businessService elements are contained by businessEntity elements and bindingTemplate elements are contained by businessService elements. Independent changes made to contained entities of such entities (for example, changes to an existing businessService within a businessEntity by means of a save_service API call) do not affect the value of the modified element associated with the containing entity. Instead, the **modifiedIncludingChildren** element in the containing entity contains the maximum of its own modified element and the modifiedIncludingChildren elements of each of the entities it contains

(if any). If a contained entity is deleted or moved elsewhere, the modifiedIncludingChildren element is also updated, since such operations would otherwise not be documented elsewhere. Changes in a service that is being projected do not affect the modifiedIncludingChildren element of the businessEntity in which it is projected. The modifiedIncludingChildren element should not be returned for operationalInfo elements corresponding to bindingTemplate or tModel elements since there are no contained elements that can be modified independently.

The degree to which the clocks of each UDDI node used to generate the created, modified, and modifiedIncludingChildren elements are synchronized is not architecturally specified, but rather is a matter of registry policy. Likewise, the frequency with which each clock is incremented (e.g.: 60Hz, 100Hz, etc.) is also a matter of registry policy.

The UDDI node (if any) that has custody of the entity to which an operationalInfo element is attached is identified by the **nodeID** element. The nodeID contains a unique key that is used to identify this node within a UDDI registry. As described in Section 7.5.2 *Configuration of a UDDI Node – operator element* for nodes that implement UDDI Replication, this element MUST match the value specified in the Replication Configuration element associated with the node.

A node may provide an indication of the owner of the data corresponding to the entityKey in the **authorizedName** element. The exact contents of this element and how the authorizedName element should be interpreted depends on the authentication, identification and privacy policies of the registry and node (see Chapter 9, *Policy*).

In a registry with multiple nodes, the operationalInfo MUST include the information required so that inquiry results are consistent across all nodes. The nodeID, authorizedName, modified and created elements are mandatory in multiple node registries both in responses to get_operationalInfo calls and in replication changeRecords containing operationalInfo elements. The modifiedIncludingChildren element MUST also be present in multiple node registries for operationalInfo elements corresponding to businessEntity and businessService elements.

4 Using UDDI APIs

UDDI specifies a number of API sets that are described in Chapter 5 *UDDI Programmer APIs* and Section 7.4 *Replication API Set*. If a node claims to support a UDDI API it MUST implement the API in conformance with this specification. Node and registry policy determine the transport and security mechanisms used for each API set. See Chapter 9 *Policy* for more information.

A UDDI registry MUST have at least one node that offers a Web service compliant Inquiry API set. A UDDI registry SHOULD have at least one node that offers a Web service compliant with the Publication, Security, and Custody and Ownership Transfer API sets. If a UDDI registry has multiple nodes, all nodes SHOULD offer Web services that are compliant with the Replication API set. The Subscription and Value Set API sets are OPTIONAL for all nodes and all registries.

The API descriptions that follow in Chapter 5 *UDDI Programmers APIs* designate input elements as optional or required. Required input elements MUST be provided within the guidelines described by the UDDI schema and in the API descriptions. Optional input elements MAY be provided, and when they are, they too must follow the guidelines described by the UDDI schema and in the API description.

4.1 SOAP Usage

This section describes the SOAP specific conventions and requirements applicable to UDDI.

Any use of SOAP by a UDDI implementation that differs from or extends the behavior described below should be modeled by publishing a tModel to represent this different use of SOAP. Any Web services that make use of the different SOAP behavior should reference the tModel in the tModelInstanceDetails of the Web service's bindingTemplate. See Section 9.4.4 *UDDI Data and Information Model* for more information.

4.1.1 Support for SOAPAction

SOAP 1.1 requires the presence of the Hyper Text Transport Protocol (HTTP) header field named SOAPAction when an HTTP binding is specified. UDDI requires the presence of this HTTP Header field to be SOAP 1.1 compliant. Different SOAP toolkits treat this HTTP header field differently. For maximum tool compatibility, the SOAPAction may contain any value, including an empty string.

Both of the following message styles (among others) are permitted in UDDI.

```
POST /someVerbHere HTTP/1.1
Host: www.somenode.org
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: ""

<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
<Body>
   <get_bindingDetail xmlns="urn:uddi-org:api_v3">
…
```

and

```
POST /someVerbHere HTTP/1.1
Host: www.somenode.org
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "get_bindingDetail"

<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
<Body>
   <get_bindingDetail xmlns="urn:uddi-org:api_v3">
…
```

4.1.2 Support for SOAP Actor

The SOAP Actor feature is not supported by UDDI. UDDI nodes MUST reject any request that arrives with a SOAP Actor attribute in the SOAP Header element by returning a generic SOAP fault with no detail element and a "Client" faultcode. The faultstring will clearly indicate the problem.

4.1.3 Support for SOAP encoding

The SOAP encoding feature (i.e., SOAP 1.1. section 5) is not supported by UDDI. In messages sent to a UDDI node there must be no claims made about the encoding style of any element within the "urn:uddi-org:*" namespace. If such claims are made, the node must respond with a generic SOAP fault with no detail element and a "Client" faultcode. The faultstring will clearly indicate the problem

4.1.4 Support for SOAP Headers

UDDI registries MAY ignore the contents of SOAP header. SOAP headers that have the must_understand attribute set to true MUST be rejected with a SOAP fault - MustUnderstand. UDDI registries MAY ignore other extension headers received.

4.1.5 Support for SOAP Fault

UDDI registries signal a generic SOAP Fault[8] when unknown API references are invoked, validation failures occur, etc. UDDI specific errors MUST be handled via a SOAP Fault element containing a UDDI dispositionReport element. The following SOAP fault codes are used:

- **VersionMismatch**: An invalid namespace reference for the SOAP envelope element was passed. The valid namespace value is "http://www.xmlsoap.org/soap/envelope/".
- **MustUnderstand**: A SOAP header element, permitted to be ignored by a UDDI node, was received with the Must_Understand attribute set to true. In response, a UDDI node MUST return this response. See Section 4.8 *Success and Error* Reporting and Chapter 12 *Error Codes.*
- **Client**: A message was incorrectly formed or did not contain enough information to perform more exhaustive error reporting.
- **Server**: The Server class of errors indicate that the message could not be processed for reasons not directly attributable to the contents of the message itself but rather to the processing of the message. For example, processing could include

[8] See section 4.4.1, "SOAP Fault Codes," in SOAP 1.1 specification for descriptive information.

communicating with an upstream processor which did not respond. The message may succeed at a later point in time.

4.1.6 XML prefix conventions – default namespace support

UDDI nodes are REQUIRED to support the use of the default namespaces (i.e. no XML prefix) in SOAP request and response documents as shown in the following HTTP example:

```
POST /someVerbHere HTTP/1.1
Host: www.example.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: ""

<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
   <Body>
        <get_bindingDetail xmlns="urn:uddi-org:api_v3">
…
```

4.2 XML Encoding Requirements

All messages sent to and received from UDDI nodes MUST be encoded in either UTF-8 or UTF-16, and MUST specify the Hyper Text Transport Protocol (HTTP) Content-Type header with a charset parameter of "utf-8" or "utf-16" respectively and a content type of text/xml. Other encoding name variants, such as UTF8, UTF_8, etc. MAY NOT be used.

All parts of the Content-type header are case insensitive and quotations are optional[9]. UDDI nodes MUST reject messages that do not conform to this requirement. For simplification purposes, all examples in this document use UTF-8.

An example of a valid HTTP Content-Type header specifying UTF-8 encoding of the message is:

```
Content-type: text/xml; charset="utf-8"
```

An example of a valid HTTP Content-Type header specifying UTF-16 encoding of the message is:

```
Content-type: text/xml; charset="utf-16"
```

4.3 Support for Unicode: Byte Order Mark

Unicode UTF-8 allows data to be transmitted with an OPTIONAL three-byte signature, also known as Byte Order Mark (BOM), preceding the XML data. This signature does not contain information that is useful for decoding the contents; but, in the case of UTF-8, tells the receiving program that the rest of the text is in UTF-8. Its presence makes no difference to the endianness of the byte stream as UTF-8 always has the same byte order. The BOM is not part of the textual content therefore UDDI nodes MAY remove the BOM prior to processing messages received.

UDDI nodes MUST be prepared to accept messages that contain Byte Order Marks, but the Byte Order Mark is not required to process SOAP messages successfully.

UDDI nodes MUST NOT return a Byte Order Mark with any of the response messages defined in this specification.

All UDDI nodes MUST support all of the Unicode characters, including all compatibility characters. See Section 4.6.1.1 *XML Normalization and Canonicalization* for further

[9] See http://www.ietf.org/rfc/rfc2045

information on required behavior with respect to character set normalization and canonicalization.

4.4 About uddiKeys

UDDI registries MUST use keys that conform to the following grammar. All UDDI keys are URIs that conform to RFC 2396 but not all URIs are valid UDDI keys. The URIs that are valid as UDDI keys correspond to a subset of the opaque part alternative of the absoluteURI rule in RFC 2396. Further, registries MUST use keys from the scheme "uddi" following the syntactic and semantic rules for that scheme as given in this section, in Section 5.2.2 *Publishing Entities with Publisher Assigned Keys*, in Section 8.2 *Data Management Policies and Procedures Across Registries*, and Section 9.4.3 *Policy Abstractions for the UDDI keying scheme*. The primary motivations for the scheme for uddiKeys is to allow publishers to specify keys for entities they publish in UDDI registries using "sensible looking" keys and to promote interoperation among UDDI registries. See Chapter 10 *Multi-Version Support* for issues regarding backwards compatibility.

Keys in UDDI are declared as language independent case insensitive and must be case-folded by nodes as part of processing any API. With the inclusion of the attribute caseMapKind="fold" from Schema Centric Canonicalization in the schema declaration for uddiKey type, the normalized form of a uddiKey element is produced using Unicode Case Folding as defined in the *Unicode Technical Report on Case Mappings* (see http://www.unicode.org/unicode/reports/tr21/).

A derivedKey has the form uddiKey ":" KSS, where the key specific string (KSS) is composed of upper and lowercase characters, numbers, and other symbols permitted in a URI. See Section 5.2.2.1 *Key generator keys and their partitions* for a description of derivedKey.

4.4.1 Key Syntax[10]

```
uddiScheme          =   %d117.100.100.105; "uddi" in lower case

uddiKey             =   nonKeyGeneratorKey / keyGeneratorKey

nonKeyGeneratorKey  =   uuidKey / domainKey / derivedKey

uuidKey             =   uddiScheme ":" uuid_part

uuid_part           =   8HEXDIG "-"
                        4HEXDIG "-"
                        4HEXDIG "-"
                        4HEXDIG "-"
                        12HEXDIG

domainKey           =   uddiScheme ":" hostname

hostname            =   *(domainlabel ".") toplabel ["."]

domainlabel         =   alphanum / alphanum *(alphanum / "-") alphanum

toplabel            =   ALPHA / ALPHA *(alphanum / "-") alphanum

alphanum            =   ALPHA / DIGIT

derivedKey          =   nonKeyGeneratorKey ":" KSS (Key Specific String)

keyGeneratorKey     =   nonKeyGeneratorKey ":" "keygenerator"

KSS                 =   1*uric ; KSS MUST NOT be "keygenerator"
```

[10] The notation used here is "Augmented Backus-Naur Form" (ABNF) as defined in RFC 2234.

```
uric            =   reserved / unreserved / escaped

reserved        =   ";" / "/" / "?" / "@" / "&" / "=" / "+" / "$" / ","

unreserved      =   alphanum / mark

mark            =   "-" / "_" / "." / "!" / "~" / "*" / "'" / "(" / ")"

escaped         =   "%"  HEXDIG HEXDIG
```

There are some extra restrictions on domain names that are not captured in the ABNF syntax above:

1. The maximum length of a string representation of a hostname is 253 characters/octets.
2. The maximum length of an individual domainlabel is 63 characters/octets.

There is an additional restriction on the Key Specific Sting (KSS) that is not captured in the ABNF syntax above:

1. KSS MUST NOT be "keygenerator".

The keyword "keygenerator" is case-insensitive.

4.4.2 Examples of keys

The following are examples of legal domainKeys.

uddi:tempuri.com

Here, "tempuri.com" is the domain of this key.

uddi:us.tempuri.com

Here, "us.tempuri.com" is the domain.

The following is an example of a legal uuidKey.

uddi:4CD7E4BC-648B-426D-9936-443EAAC8AE23

"4CD7E4BC-648B-426D-9936-443EAAC8AE23" is the uuid of this key.

The following are examples of legal derivedkeys

uddi:AC104DCC-D623-452F-88A7-F8ACD94D9B2B:xyzzy

This is a derived Key based on the <uuidKey> "uddi:AC104DCC-D623-452F-88A7-F8ACD94D9B2B". The string "xyzzy" is key's KSS.

uddi:tempuri.com:fish:buyingservice

This key is based on the derivedKey "uddi:tempuri.com:fish". The string "buyingService" is the key's KSS.

The following is an example of a legal key generator key

uddi:tempuri.com:keygenerator

This key is based on the domainKey "uddi:tempuri.com"

4.5 Data insertion and document order

4.5.1 Inserting Data in Entities During save_xx Operations

When saving a businessEntity, businessService, bindingTemplate or tModel, the UDDI node is required to add or replace certain elements and attributes if they are not present or are incorrectly specified in the entity passed to the save_xx API. These are: For businessEntity, businessService, bindingTemplate and tModel structures, the businessKey, serviceKey, bindingKey, and tModelKey of the structure being saved.

4.5.2 Inserting Elements in Existing Entities

When a new child element is inserted by a publication API, the UDDI node MUST add the new child as the last of its siblings. For example, the save_service call can be used to add a businessService to a businessEntity. The added businessService appears as the last one in the (possibly single item) list of such businessService structures. When inserting a businessService using save_service or a bindingTemplate using save_binding, any digital signatures on the containing UDDI data structure may become invalid with the addition of a new child.

4.5.3 Preservation of Document Order

The UDDI data model requires UDDI nodes to preserve the order of all descendent elements in the UDDI core data structures. When a UDDI node responds to an inquiry API call, the descendent elements of the core data structures in the response must have the order specified by their publishers or by the order of insertion.

Preservation of document order in UDDI implies that all elements in a sequence MUST be preserved. It is a requirement not to de-duplicate elements of a sequence, other than for keyed entities as described in sections 5.2.16.4 *save_business return section* and 5.2.17.4 *save_service return* section.

4.6 XML Normalization and Canonicalization

UDDI registries provide publishers with the ability to digitally sign and save entities they publish, and inquirers with the ability to retrieve and validate the digital signatures on published material. In order for this to be possible, publishers and registries MUST handle "normalization" and "canonicalization" as described in this section.

Normalization is the process of standardizing the representation of the characters that make up a document. In Unicode data there is often more than one way to represent a given glyph. For example, the character "Å" may be represented as one single character "LATIN CAPITAL LETTER A WITH RING ABOVE" (hexadecimal 00C5), as another single character "ANGSTROM SIGN" (hexadecimal 212B) or as a composition of "LATIN CAPITAL LETTER A" (hexadecimal 0041) and "COMBINING RING ABOVE" (hexadecimal 030A). Normalization chooses one standard representation in every such case.

Canonicalization is the process of generating a standard representation of XML. It deals with issues such as the representation of tags; attribute ordering; namespace declaration, expansion and ordering; and whitespace handling.

4.6.1 Behavior of UDDI nodes

4.6.1.1 Normalization and Canonicalization

UDDI registries MUST exhibit certain behavior with respect to the saved vs. retrieved representations of the entities they handle. Aspects of this behavior REQUIRE attention to the *Schema Centric XML Canonicalization* (see

http://uddi.org/pubs/SchemaCentricCanonicalization.htm) for this function. More specifically, registries MUST exhibit the following behavior with respect to the data they store and retrieve. Let:

- C(d, S) be the *Schema Centric XML Canonicalization* transform of document d with respect to the set of schemas S,
- U be the set of UDDI v3 schemas,
- x and y be UDDI entities,
- R be a UDDI registry.

For all x saved in R, if y is x as retrieved from R, it MUST be the case that C(x, U) = C(y, U) in a literal bit-by-bit sense.

Stated informally, if you save an entity in a UDDI registry and later retrieve it, the canonicalization of what you saved will be the same as the canonicalization of what you got back. However, this is only guaranteed to be true with respect to the Schema Centric Canonicalization algorithm; in particular such guarantees are not provided with respect to the Canonical XML algorithm or its Exclusive Canonical XML variation (see http://www.w3.org/TR/xml-exc-c14n).[11]

4.6.2 Client Behavior

The behavior of UDDI registries with respect to normalization and canonicalization means that if an entity, x, is published and later retrieved from a registry as y, y will not, in general, be precisely the same bits as x; only a canonicalized form of x and y are guaranteed to be bitwise identical. This behavior means that for digital signatures to work, publishers and inquirers SHOULD take certain actions.

4.6.2.1 Publishers

Publishers SHOULD prepare entities they wish to sign by including in their XML DSIG SignedInfo a Transform which canonicalizes them using *Schema Centric XML Canonicalization* (see Section 4.6.1.1 *Normalization and Canonicalization*) before calculating the signatures. Publishers SHOULD avoid inserting elements into published signed entities as doing so likely invalidates the signature.

4.6.2.2 Inquirers

To validate signed entities, inquirers SHOULD adhere to the strictures and processes of the XML DSIG specification. If, as will almost always be the case in UDDI, the *Schema Centric Canonicalization*[12] algorithm was indicated by the signer, then execution of the algorithm will be necessary as part of the process of validating the signature.

4.7 About Access Control and the authInfo Element

The Authorization Policy for a Registry defines how/if access control is implemented. See Chapter 9 for a discussion of Policy issues.

The authInfo element is an OPTIONAL element on every API call of the Publication, Inquiry and Subscription API sets. Using an optional element allows different UDDI registries and

[11] As a matter of implementation, registries can straightforwardly support this guarantee by doing little more than simply returning data which is valid against its schema(s). That is, the implementation burden of this requirement is minimal. In particular, registries need not actually execute the canonicalization algorithm as part of the save or retrieval processes.

[12] http://uddi.org/pubs/SchemaCentricCanonicalization.htm

nodes within the registries to implement access control on whichever sets of operations such control is desired.

AuthInfo is an opaque element whose content is meaningful only to the node that created it. It is intended to enable a variety of authentication mechanisms. For example, it may be used with:

- Id/password based systems in which the authInfo is an authorization token generated by the authentication operation (i.e. Kerberos Tickets)
- Authorization assertions (i.e., SAML, X509 Attribute Certificates)

When a node uses authInfo elements it MAY offer the get_authToken and discard_authToken APIs as a means of obtaining and disposing of them. Alternatively, or in addition, it MAY offer other means for doing this.

The use of authInfo elements is not the only means a node may use for access control. For example, if a node chooses to implement authentication at the transport level, it may well rely on the authorization information supplied by the transport.

4.8 Success and Error Reporting

The first line of error reporting is governed by the SOAP specification. SOAP fault reporting and fault codes will be returned for most invalid requests or any request where the intent of the caller cannot be determined.

If any application level error occurs in processing a request message, a dispositionReport element will be returned to the caller within a SOAP fault report. Faults that contain disposition reports contain error information that includes descriptions and the type of key associated with an entity that can be used to determine the cause of the error. API-specific interpretations of error codes are provided with each API description.

In a manner consistent with the SOAP processing rules (Section 6.2 of the SOAP 1.1 specification) UDDI follows the semantics of the Hyper Text Transport Protocol (HTTP) Status codes for communicating status information in HTTP. As is the case for SOAP, success reporting will use a 200 status code to indicate that the client's request including the SOAP component was successfully processed.

UDDI application-level errors SHOULD be conveyed using standard HTTP status code where a 500-level code indicates a server-induced error. In such cases, the UDDI node MUST issue an HTTP 500 "Internal Server Error" response and return a dispositionReport inside a SOAP fault report.

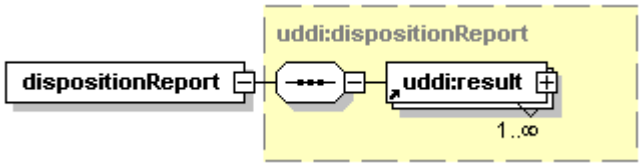
Many of the API constructs defined in this specification allow one or more of a given type of information to be passed. These API calls each conceptually represent a request on the part of the caller. The general error handling treatment recommended for UDDI nodes is to detect errors in a request prior to processing the request. Any errors in the request detected will invalidate the entire request, and cause a dispositionReport to be generated within a SOAP Fault as described below.

In the case of an API call that involves passing multiples of a given structure, the dispositionReport will call out only the first detected error, and is not responsible for reporting multiple errors or reflecting intermediate "good" data. In situations where a specific reference within a request causes an error to be generated, the corresponding disposition/fault report will contain a clear indication of the key value that caused the rejection of the rejected request.

In general, UDDI nodes may return any UDDI error code needed to describe an error. The error codes specified within each API call description are characteristic of the API call, but other UDDI error codes may be returned in unusual circumstances or when doing so adds additional descriptive information. See Chapter 12 *Error Codes* for a summary of UDDI error codes.

4.8.1 dispositionReport element

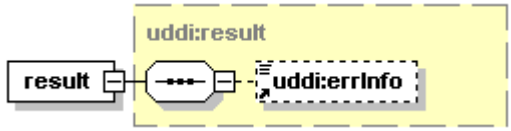
Error information is always returned in the dispositionReport. A dispositionReport has the form:



Attributes

Name	Use
truncated	optional

The **dispositionReport** is a non-empty list of error conditions, each described in a **result** element. The **truncated** attribute indicates whether error conditions occurred that are not listed in the dispositionReport.



Attributes

Name	Use
keyType	optional
errno	required

The result element contains an optional **keyType** and a required **errno** attribute. The errno attribute is set to the value described in Chapter 12 *Error Codes*. The keyType attribute is used to indicate the type of the uddiKey that is being reported on, e.g. in an E_invalidKeyPassed error condition. Valid values for keyType are "businessKey", "serviceKey", "bindingKey", "tModelKey" and "subscriptionKey". Detailed information about the error condition can be found in the optional **errInfo** element.

The errInfo element, if necessary, describes the error condition in more detail. It contains a string that is adorned with an **errCode** attribute, set to the string described in Chapter 12 *Error Codes*.

Like other UDDI data structures, the disposition report includes a namespace that identifies the UDDI version for which it applies. When a UDDI node receives a message with a namespace that cannot be used to determine the version, a disposition report is return for the most current UDDI version that the node supports.

4.8.2 Error reporting using the dispositionReport element

All application errors are communicated via the use of the SOAP FAULT element. The general form of an error report is:

```
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoaporg.org/soap/envelope/">
   <Body>
     <Fault>
       <faultcode>Client</faultcode>
       <faultstring>Client Error</faultstring>
       <detail>
         <dispositionReport xmlns="urn:uddi-org:api_v3">
            <result errno="10500">
               <errInfo errCode="E_fatalError">The findQualifier
                  value passed is unrecognized: XYZ</errInfo>
            </result>
         </dispositionReport>
       </detail>
     </Fault>
   </Body>
```

```
</Envelope>
```

Multiple *result* elements may be present within the dispositionReport element, and can be used to provide very detailed error reports for multiple error conditions. The number of *result* elements returned within a disposition report is implementation specific. In general it is permissible to return an error response as soon as the first error in a request is detected. References within the API reference sections that describe error text content rules pertain to the content of the errInfo element.

5 UDDI Programmers APIs

This API reference is divided into a number of logical sections, each addressing a particular programming focus. These sections cover the inquiry API, the publishing API and the OPTIONAL security, custody transfer, subscription and value set APIs.

In all cases, the XML structures, attributes and element names shown in the API examples are derived from the UDDI API schemas described in Chapter 2 *UDDI Schemas*. For a full understanding of structure contents, refer to this chapter, the UDDI schemas, and Chapter 3 *UDDI Registry Data Structures*.

Each API set has one or more corresponding tModels that are referenced in bindingTemplate structures to indicate that a compliant Web service is offered for the API set. See Section 11.1.9 *UDDI Registry API tModels*.

5.1 Inquiry API Set

The inquiry API set allows one to locate and obtain detail on entries in a UDDI registry. The Inquiry API provides three forms of query that follow broadly used conventions which match the needs of software traditionally used with registries. Three distinct patterns of inquiry are supported.

5.1.1 The browse pattern

Software that allows people to explore and examine large quantities of data requires browse capabilities. The browse pattern characteristically involves starting with some broad information, performing a search, finding general result sets and then selecting more specific information for drill-down.

This specification supports the browse pattern by way of the API calls involving "find" operations (hereafter referred to as the "find_xx" APIs). These calls form the search capabilities provided by the API and are matched with summary return structures that return overview information about the registered information associated with the inquiry API and the search criteria specified in the inquiry.

A typical browse sequence might involve finding whether a particular business with a particular name has any information registered. This sequence starts with a call to find_business, passing the business name (which could involve the use of just a portion of the name together with a wildcard character by using the approximateMatch findQualifier described below). This returns a businessList result - overview information (keys, names and descriptions) derived from the registered businessEntity information, matching on the name. Information in the list may be used to select among the businesses and then to drill down into the corresponding businessService information, looking for one which matches a particular technical fingerprint (i.e., tModel such as for purchasing, shipping, etc) using the find_service API call. UDDI provides many similar sequences of API calls that let callers start with a broad notion of the kind of information they wish to retrieve from a registry, retrieve summary information, and then drill down to get details.

5.1.2 The drill-down pattern

Each instance of the core data structures – businessEntity, businessService, bindingTemplate and tModel – has a key which is one of the items in the summary information retrieved by

find_xx APIs. Given such a key, it is easy to retrieve the full registered details for the corresponding instance by passing the key to the relevant get_xx API.

Continuing the example from the previous section on browsing, the businessKey associated with the business being sought is one of the items in the businessList returned by find_business. This key can be passed as an argument to get_businessDetail. Upon success, this API returns a businessDetail containing the full registered information, including the businessEntity structure for the entity whose key value was passed.

5.1.3 The invocation pattern

To have an application take advantage of a Web service that is registered within a UDDI registry, that application must be prepared to use the information found in the registry for the specific Web service being invoked. This type of inter-business service call has traditionally been a task that is undertaken entirely at development time. The degree to which this changes with Web services is an application design choice, but the existence of UDDI registry entries makes it significantly easier to do dynamic binding using the following pattern.

Obtain the bindingTemplate data for the Web service of interest from a UDDI registry, such as the UDDI Business Registry. Typically this is done using one of the browse-and-drill-down patterns discussed above. The bindingTemplate contains the specific details about an instance of a given interface type, including the location at which a program starts interacting with the Web service. The calling application caches this information and uses it to contact the Web service at the registered address whenever it needs to communicate with the Web service instance.

If a call fails using cached information previously obtained from a UDDI registry, the application SHOULD query the UDDI registry for fresh bindingTemplate information. The proper call is get_bindingDetail passing the original bindingKey value. If the data returned is different from the cached information, the application SHOULD retry the invocation using the fresh information. If the result of this retry is successful, the new information SHOULD replace the cached information.

By using this pattern with Web services, applications can interact with partners without undue communication and coordination costs. For example, if a business has activated a disaster recovery site, most of the calls from partners will fail when they try to invoke Web services at the failed site. By updating the UDDI information with the new address for the Web service, partners who use the invocation pattern will automatically locate the new Web service information and recover without further administrative action. Cached binding information could alternatively be kept up to date by means of notification or polling.

5.1.4 Find Qualifiers

Each of the find_xx APIs accepts an optional findQualifiers argument, which may contain multiple findQualifier values. Find qualifiers may be either tModelKeys or may be referenced by a string containing a "short name". Each of the pre-defined findQualifiers in UDDI can be referenced using either the appropriate tModelKey, or by its short name. Registries MUST support both forms, and MUST accept the find qualifiers case-insensitively. The use of tModelKeys for findQualifiers allows extension to create additional new qualifiers, but registries are not obligated to support them. Find qualifiers not recognized by a node will return the error E_unsupported.

Matching behavior for the find_xx APIs when multiple criteria are specified is logical "AND" by default. Find qualifiers allow the default search behaviors to be overridden. Not all find_xx APIs support all findQualifier element values. The following table identifies which findQualifiers apply to each API:

Table 1: Find Qualifiers by API

Find Qualifier Short Name and tModel Name	find_business	find _service	find_binding	find_tModel	find_related Businesses
"**andAllKeys**" (uddi-org:andAllKeys)	YES default for categoryBag, tModelBag, applicable to identifierBag	YES default for tModelBag & categoryBag	YES default for categoryBag, tModelBag	YES default for categoryBag, applicable to identifierBag	NO
"**approximateMatch**" (uddi-org:approximateMatch:SQL99)	YES	YES	YES	YES	YES
"**binarySort**" (uddi-org:binarySort)	YES	YES	NO	YES	YES
"**bindingSubset**" (uddi-org:bindingSubset)	YES applicable to categoryBag on bindingTemplate	YES applicable to categoryBag on bindingTemplate	NO	NO	NO
"**caseInsensitiveSort**" (uddi-org:caseInsensitiveSort)	YES	YES	NO	YES	YES
"**caseInsensitiveMatch**" (uddi-org:caseInsensitiveMatch)	YES	YES	YES	YES	YES
"**caseSensitiveSort**" [13] (uddi-org:caseSensitiveSort)	YES	YES	NO	YES	YES
"**caseSensitiveMatch**" [14] (uddi-org:caseSensitiveMatch)	YES	YES	YES	YES	YES
"**combineCategoryBags**" (uddi-org:combineCategoryBags)	YES	YES	NO	NO	NO
"**diacriticInsensitiveMatch**" [15] (uddi-org:diacriticInsensitiveMatch)	YES	YES	YES	YES	YES
"**diacriticSensitiveMatch**" [16] (uddi-org:diacriticSensitiveMatch)	YES	YES	YES	YES	YES

[13] This is the default behavior.

[14] This is the default behavior.

[15] Implementation of this findQualifier by nodes is OPTIONAL.

[16] This is the default behavior.

Find Qualifier Short Name and tModel Name	find_business	find _service	find_binding	find_tModel	find_related Businesses
"**exactMatch**" [17] (uddi-org:exactMatch)	YES	YES	YES	YES	YES
"**signaturePresent**" (uddi-org:signaturePresent)	YES	YES	YES	YES	YES
"**orAllKeys**" (uddi-org:orAllKeys)	YES default for identifierBag, applicable to tModelBag or categoryBag	YES applicable to categoryBag, tModelBag	YES applicable to categoryBag, tModelBag	YES applicable to categoryBag, default for identifierBag	NO
"**orLikeKeys**" (uddi-org:orLikeKeys)	YES applicable to identifierBag, categoryBag	YES applicable to categoryBag	YES applicable to categoryBag	YES applicable to categoryBag, identifierBag	NO
"**serviceSubset**" (uddi-org:serviceSubset)	YES	NO	NO	NO	NO
"**sortByNameAsc**" [18] (uddi-org:sortByNameAsc)	YES	YES	NO	YES	YES
"**sortByNameDesc**" (uddi-org:sortByNameDesc)	YES	YES	NO	YES	YES
"**sortByDateAsc**" (uddi-org:sortByDateAsc)	YES	YES	YES default behavior	YES	YES
"**sortByDateDesc**" (uddi-org:sortByDateDesc)	YES	YES	YES	YES	YES
"**suppressProjectedServices**" (uddi-org:suppressProjectedServices)	YES	YES	NO	NO	NO
"**UTS-10**" [19] (uddi-org:UTS-10)	YES	YES	NO	YES	YES

5.1.4.1 Invalid Find Qualifier Combinations

Using a findQualifier with one of the find_xx APIs to which it does not apply, will generally result in that qualifier being ignored, but there are a few situations for which certain

[17] This is the default behavior.

[18] This is the default behavior.

[19] Implementation of this findQualifier by nodes is OPTIONAL.

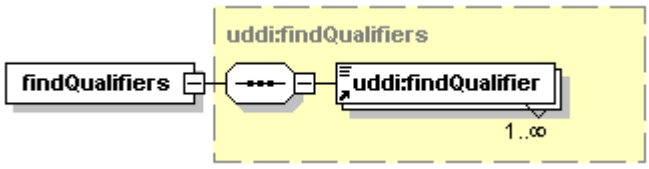
findQualifiers elements are mutually exclusive and supplying them together will result in an E_invalidCombination error. The invalid combinations are:

- andAllKeys, orAllKeys, and orLikeKeys are mutually exclusive
- sortByNameAsc and sortByNameDesc are mutually exclusive
- sortByDateAsc and sortByDateDesc are mutually exclusive
- combineCategoryBags, serviceSubset and bindingSubset are mutually exclusive
- exactMatch and approximateMatch are mutually exclusive
- exactMatch and caseInsensitiveMatch are mutually exclusive
- binarySort and UTS-10 are mutually exclusive, as are all collation algorithm tModels with each other
- diacriticSensitiveMatch and diacriticInsensitiveMatch are mutually exclusive
- exactMatch and diacriticInsensitiveMatch are mutually exclusive
- caseSensitiveSort and caseInsensitiveSort are mutually exclusive
- caseSensitiveMatch and caseInsensitiveMatch are mutually exclusive

See Chapter 11, *Utility tModels and Conventions* for further information on find qualifier tModels.

5.1.4.2 General Form of Find Qualifiers

Find qualifiers are expressed by using a findQualifiers argument. The general form of the findQualifiers element is:



where a findQualifier can be either a string (with a maximum length of 255), or a tModelKey.

5.1.4.3 Find Qualifier Descriptions

The value passed in each findQualifier element indicates the behavior change desired. This list defines the set of UDDI defined valid qualifiers. Nodes MUST implement all of these except as noted.

- andAllKeys: this changes the behavior for identifierBag to AND keys rather than OR them. This is already the default for categoryBag and tModelBag.
- approximateMatch: signifies that wildcard search behavior is desired. This behavior is defined by the uddi-org:approximatematch:SQL99 tModel and means "approximate matching as defined for the character like predicate in *ISO/IEC9075-2:1999(E) Section 8.5 like predicate*, where the percent sign (%) indicates any number of characters and an underscore (_) indicates any single character. The backslash character (\) is used as an escape character for the percent sign, underscore and backslash characters. This find qualifier adjusts the matching behavior for name, keyValue and keyName (where applicable).
- binarySort: this qualifier allows for greater speed in sorting. It causes a binary sort by name, as represented in Unicode codepoints.

- bindingSubset: this is used in the find_business API or the find_service API and is used only in conjunction with a categoryBag argument. It causes the component of the search that involves categorization to use only the categoryBag elements from contained bindingTemplate elements within the registered data and ignores any entries found in the categoryBag which are not direct descendent elements of registered businessEntity elements or businessService elements. The resulting businessList or serviceList return those businesses or services that matched based on this modified behavior, in conjunction with any other search arguments provided. Additionally, in the case of the returned businessList from a find_business, the contained serviceInfos elements will only reflect summary data (in a serviceInfo element) for those services (contained or referenced) that contained a binding that matched on one of the supplied categoryBag arguments.
- caseInsensitiveMatch: signifies that the matching behavior for name, keyValue and keyName (where applicable) should be performed without regard to case.
- caseInsensitiveSort: signifies that the result set should be sorted without regard to case. This overrides the default case sensitive sorting behavior.
- caseSensitiveMatch: signifies that the matching behavior for name, keyValue and keyName (where applicable) should be performed with regard to case. This is the default behavior.
- caseSensitiveSort: signifies that the result set should be sorted with regard to case. This is the default behavior.
- combineCategoryBags: this may only be used in the find_business and find_service calls. In the case of find_business, this qualifier makes the categoryBag entries for the full businessEntity element behave as though all categoryBag elements found at the businessEntity level and in all contained or referenced businessService elements and bindingTemplate elements were combined. Searching for a category will yield a positive match on a registered business if any of the categoryBag elements contained within the full businessEntity element (including the categoryBag elements within contained or referenced businessService elements or bindingTemplate elements) contains the filter criteria. In the case of find_service, this qualifier makes the categoryBag entries for the full businessService element behave as though all categoryBag elements found at the businessService level and in all contained or referenced elements in the bindingTemplate elements were combined. Searching for a category will yield a positive match on a registered service if any of the categoryBag elements contained within the full businessService element (including the categoryBag elements within contained or referenced bindingTemplate elements) contains the filter criteria. This find qualifier does not cause the keyedReferences in categoryBags to be combined with the keyedReferences in keyedReferenceGroups in categoryBags when performing the comparison. The keyedReferences are combined with each other, and the keyedReferenceGroups are combined with each other.
- diacriticInsensitiveMatch: signifies that matching behavior for name, keyValue and keyName (where applicable) should be performed without regard to diacritics. Support for this findQualifier by nodes is OPTIONAL.
- diacriticSensitiveMatch: signifies that the matching behavior for name, keyValue and keyName (where applicable) should be performed with regard to diacritics. This is the default behavior.
- exactMatch: signifies that only entries with names, keyValues and keyNames (where applicable) that exactly match the name argument passed in, after normalization, will be returned. This qualifier is sensitive to case and diacritics where applicable. This qualifier represents the default behavior.

- orAllKeys: this changes the behavior for tModelBag and categoryBag to OR the keys within a bag, rather than to AND them. Using this findQualifier with both a categoryBag and a tModelBag, will cause all of the keys in BOTH the categoryBag and the tModelBag to be OR'd together rather than AND'd. It is not possible to OR the categories and retain the default AND behavior of the tModels. The behavior of keyedReferenceGroups in a categoryBag is analogous to that of individual keyedReferences, that is, the complete categoryBag is changed to OR the keys.
- orLikeKeys: when a bag container (i.e. categoryBag or identifierBag) contains multiple keyedReference elements, any keyedReference filters that come from the same namespace (e.g. have the same tModelKey value) are OR'd together rather than AND'd. For example "*find any of these four values from this namespace, and any of these two values from this namespace*". The behavior of keyedReferenceGroups is analogous to that of keyedReferences, that is, keyedReferenceGroups that have the same tModelKey value are OR'd together rather than AND'd.
- serviceSubset: this is used only with the find_business API and is used only in conjunction with a categoryBag argument. It causes the component of the search that involves categorization to use only the categoryBag elements from contained or referenced businessService elements within the registered data and ignores any entries found in the categoryBag which are not direct descendent elements of registered businessEntity elements. The resulting businessList structure contains those businesses that matched based on this modified behavior, in conjunction with any other search arguments provided. Additionally, the contained serviceInfos elements will only reflect summary data (in a serviceInfo element) for those services (contained or referenced) that matched on one of the supplied categoryBag arguments.
- signaturePresent: this is used with any find_xx API to restrict the result set to entities which either contain an XML Digital Signature element, or are contained in an entity which contains one. The Signature element is retrieved using a get_xx API call and SHOULD be verified by the client. A UDDI node may or may not verify the signature and therefore use of this find qualifier, or the presence of a Signature element SHOULD only be for the refinement of the result set from the find_xx API and SHOULD not be used as a verification mechanism by UDDI clients.
- sortByNameAsc: causes the result set returned by a find_xx or get_xx inquiry APIs to be sorted on the name field in ascending order. This sort is applied prior to any truncation of result sets. It is only applicable to queries that return a name element in the top-most detail level of the result set and if no conflicting sort qualifier is specified, this is the default sorting direction. This findQualifier takes precedence over sortByDateAsc and sortByDateDesc qualifiers, but if a sortByDateXxx findQualifier is used without a sortByNameXxx qualifier, sorting is performed based on date with or without regard to name.
- sortByNameDesc: causes the result set returned by a find_xx or get_xx inquiry call to be sorted on the name field in descending order. This sort is applied prior to any truncation of result sets. It is only applicable to queries that return a name element in the top-most detail level of the result set. This is the reverse of the default sorting direction. This findQualifier takes precedence over sortByDateAsc and sortByDateDesc qualifiers but if a sortByDateXxx findQualifier is used without a sortByNameXxx qualifier, sorting is performed based on date with or without regard to name.
- sortByDateAsc: causes the result set returned by a find_xx or get_xx inquiry call to be sorted based on the most recent date when each entity, or any entities they contain, were last updated, in ascending chronological order (oldest are returned first). When names are included in the result set returned, this find qualifier may also be used in

conjunction with either the sortByNameAsc or sortByNameDesc findQualifiers. When so combined, the date-based sort is secondary to the name-based sort (results are sorted within name by date, oldest to newest).

- sortByDateDesc: causes the result set returned by a find_xx or get_xx inquiry call to be sorted based on the most recent date when each entity, or any entities they contain, were last updated, in descending chronological order (most recently changed are returned first. When names are included in the result set returned, this find qualifier may also be used in conjunction with either the sortByNameAsc or sortByNameDesc find qualifiers. When so combined, the date-based sort is secondary to the name-based sort (results are sorted within name by date, newest to oldest).
- suppressProjectedServices: signifies that service projections MUST NOT be returned by the find_service or find_business APIs with which this findQualifier is associated. This findQualifier is automatically enabled by default whenever find_service is used without a businessKey.
- UTS-10: this is used to cause sorting of results based on the Unicode Collation Algorithm on elements normalized according to Unicode Normalization Form C. When this qualifier is referenced, a sort is performed according to the Unicode Collation Element Table in conjunction with the Unicode Collation Algorithm on the name field, normalized using Unicode Normalization Form C. Support of this findQualifier by nodes is OPTIONAL.

At this time, these are the only UDDI find qualifiers defined. UDDI registries and individual nodes may define more find qualifier values than these – but all nodes and fully compatible software MUST support the above qualifiers, except where indicated otherwise.

5.1.4.4 Sorting Details

Sorting behavior of results returned as part of a UDDI inquiry is controlled by the following sort order find qualifiers: sortByDateAsc, sortByDateDesc, sortByNameAsc, sortByNameDesc, caseInsensitiveSort, binarySort and UTS-10. These find qualifiers specify four aspects of sorting behavior as shown in Table 2: Find Qualifier Sorting Behaviors below. For information on which find qualifiers are mutually exclusive, see Section 5.1.4.1 *Invalid Find Qualifier Combinations*. Not all aspects of sorting are controlled through use of a single sort order find qualifier. In order to control any combination of aspects of sorting behavior, multiple sort order find qualifiers can be specified. For example, specifying sortByNameDesc and UTS-10 causes sorting of the result set on the name element according to the Unicode Technical Standard (UTS) #10 Collation Sequence, but in descending order.

Table 2: Find Qualifier Sorting Behaviors

Find Qualifier	Field being sorted	Direction of sort	Indicates Collation sequence	Controls Case Sensitivity
sortByNameAsc	Name	Asc		
sortByNameDesc	Name	Desc		
caseInsensitiveSort				√
caseSensitiveSort				√
binarySort			√	
UTS-10			√	
sortByDateAsc	Date	Asc		
sortByDateDesc	Date	Desc		

The default sort order aspects are to perform a case sensitive sort on the primary name element (where present), or the last change date (when a name is not present), in ascending order, using the collation sequence as determined by node policy. Nodes MAY choose to perform a secondary date or name-based sort of duplicate entries in each of these cases. If a name-based findQualifier is specified without a date-based sort, then nodes MAY perform a secondary date-based sort of duplicate entries. Similarly, when a date-based sort findQualifier is specified without a name-based sort, nodes MAY perform a secondary name-based sort of duplicate entries (where applicable).

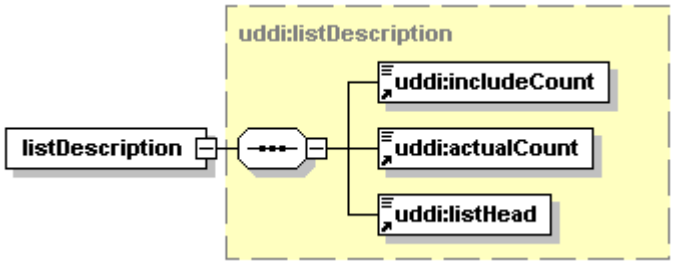
Comparison and sorting is performed based on a canonicalized representation. Specifying an unsupported sort order will result in the error E_unsupported. For more on canonicalization, refer to Section 0 *XML Normalization and Canonicalization*.

Different sorting behavior can be obtained through the use of different sort orders, which are represented by their corresponding tModels. The use of alternative collation sequences is achieved by referencing the corresponding tModelKey as the findQualifier argument supplied in the search. Support for sorting based tModels describing any collation sequences other than binary by a node is OPTIONAL.

When a result set is being sorted by name only, then by default the first name stored for the businessEntity is the one against which sorting is performed. Nodes that offer language-specific sort collation sequences MAY sort based the name element associated with the collation language.

5.1.5 Use of listDescription

Several of the inquiry APIs cause a list of results to be returned. In such cases, an element called listDescription MAY also be returned:



Where:

- ***includeCount***: is the number of list items returned for the particular response
- ***actualCount***: is the number of all available matches at the time this particular query was made
- ***listHead***: is an index (with origin of 1) which indicates the index position within all available matches of the first element of the returned result set after any sorting has been applied.

When a listDescription is returned as part of the result set, it includes a listHead value that indicates the index position of the first result in the set relative to the beginning of the list of all matches for the query. The query can specify that the result set returned should start with a particular element in the list of all matches for the query by including a listHead attribute on the query (find_xx API). The maximum number of results included in response to a query may be determined by the maxRows attribute of the query (find_xx API) or by the node's policy. The listDescription on the response is useful when the node determines that the size of the resultant list is too large to be returned or when the maxRows and listHead values specified by the client do not allow all of the results to be returned in response to a single query.

For example, a query with a maxRows attribute set to 10 could be issued to a node where 18 results match the query. The response to this query should contain items 1 through 10 and the listDescription would have an actualCount value of 18, a listHead value of 1 and an includeCount value of 10. If the data matching the query does not change and the query is sent again to the node with a listHead attribute value of 11, the result set should contain items 11-18 and the listDescription would have an actualCount value of 18, a listHead value of 11 and an includeCount value of 8. If the listHead value is less than 1, a value of 1 will be used to produce the result. If the listHead value exceeds the total number of results provided, an empty result set will be returned.

listDescription is not a true "cursoring" feature. Since both the registry content and the associated result set can change between queries, supplying a particular value for listHead on subsequent queries may result in either duplicate reporting of an element which was returned as part of the original query, or a failure to report on an element.

The results of using a find_xx API will include a listDescription only if the resultant list is greater than what a node implementation can return in a single group. For example, if the result set contains 20 items and all 20 are returned at once, then the listDescription element is allowable, but not required. If the result set is 1000 and only 500 items can be returned at once, then a listDescription is required (if the truncated attribute is not used).

5.1.6 About wildcards

The default behavior of UDDI with respect to matching is "exact match". No wildcard behavior is assumed. Many UDDI inquiry APIs take the arguments "name," "keyName," and "keyValue" whose values are of type string. All such arguments may be specified using a wildcard character to obtain an "approximate match". In order to obtain wildcard searching behavior, the findQualifier tModel uddi-org:approximateMatch:SQL99 (whose tModelKey is uddi:uddi.org:findqualifier:approximatematch), or its short name "approximateMatch" MUST be specified.

Wildcards, when they are allowed, may occur at any position in the string of characters that constitutes the argument value and may occur more than once. Wildcards are denoted with a percent sign (%) to indicate any value for any number of characters and an underscore (_) to indicate any value for a single character. The backslash character (\) is used as an escape character for the percent sign, underscore and backslash characters. Use of the "exactMatch" findQualifier will cause wildcard characters to be interpreted literally, and as such should not also be combined with the escape character. Detailed rules for interpretation are defined by the above tModel for approximate matching. Examples of the use of wildcards may be found in Appendix G *Wildcards*.

5.1.7 Matching Rules for keyedReferences and keyedReferenceGroups

When determining matching behavior in searches involving keyedReferences in categoryBags and identifierBags, a match occurs if and only if:

1. The tModelKeys refer to the same tModel. This key MUST be specified and MUST NOT be empty.
2. The keyValues match (an exact match is the default, but the matching behavior is modified appropriately if the caseInsensitiveMatch, diacriticInsensitiveMatch or approximateMatch findQualifiers are used); and
3. If the tModelKey involved is "uddi:uddi-org:general_keywords", the keyName must match (wildcard matching rules apply if the approximateMatch findQualifier is used). Omitted keyNames are treated as empty keyNames. Otherwise, keyNames are not significant unless so indicated in the individual API sections below.

A given keyedReferenceGroup "X" (e.g., within a given categoryBag) matches a keyedReferenceGroup "Y" in the registry if and only if the tModelKey assigned to the keyedReferenceGroup X matches the tModelKey assigned to the keyedReferenceGroup Y and the set of keyedReferences in "X" are a subset of the set of keyedReferences in "Y." The order of individual keyedReferences within a keyedReferenceGroup is not important. Matching rules for the individual contained keyedReference elements are the same as above.

For additional information and examples, refer to Appendix E *Using Identifiers* and Appendix F *Using Categorization.*

5.1.8 Inquiry API functions

The APIs in this section represent inquiries that can be used to retrieve data from a UDDI registry. These calls all behave synchronously and are suitable for being exposed via HTTP-POST. The calls constituting the UDDI inquiry API are:

- find_binding: Used to locate bindings within or across one or more registered businessServices. Returns a bindingDetail structure. See Section 5.1.9 *find_binding*.
- find_business: Used to locate information about one or more businesses. Returns a businessList structure. See Section 5.1.10 *find_business*.

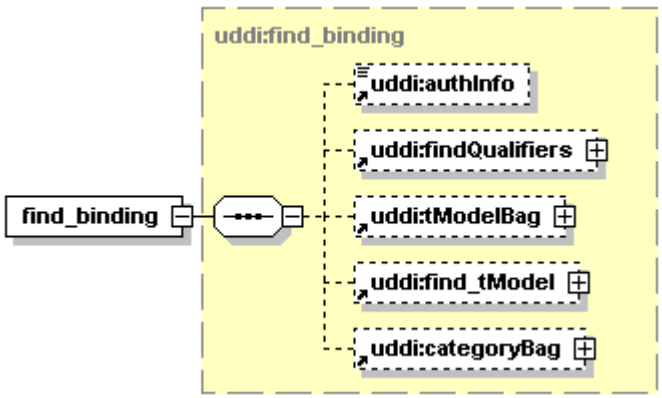
- find_relatedBusinesses: Used to locate information about businessEntity registrations that are related to a specific business entity whose key is passed in the inquiry. See Section 5.1.11 *find_relatedBusinesses*.
- find_service: Used to locate specific services within registered business entities. Returns a serviceList structure. See Section 5.1.12 *find_service*.
- find_tModel: Used to locate one or more tModel information structures. Returns a tModelList structure. See Section 5.1.13 *find_tModel*.
- get_bindingDetail: Used to get bindingTemplate information suitable for making service requests. Returns a bindingDetail structure. See Section 5.1.14 *get_bindingDetail*.
- get_businessDetail: Used to get the businessEntity information for one or more businesses or organizations. Returns a businessDetail structure. See Section 5.1.15 *get_businessDetail*.
- get_operationalInfo: Used to retrieve operational information pertaining to one or more entities in the registry. Returns an operationalInfos structure. See Section 5.1.16 *get_operationalInfo*.
- get_serviceDetail: Used to get full details for a given set of registered businessService data. Returns a serviceDetail structure. See Section 5.1.16 *get_serviceDetail*.
- get_tModelDetail: Used to get full details for a given set of registered tModel data. Returns a tModelDetail structure. See Section 5.1.18 *get_tModelDetail*.

Several of the find_xx APIs (find_binding, find_business and find_service) support nested queries, where one or more of the arguments to these APIs can itself be another (inner) query, the results of which are used to filter the overall results of the primary (outer) query along with the other criterions supplied. Any findQualifier arguments used only apply directly to the part of the query (outer or inner) for which they are supplied. They do not propagate inward or outward.

5.1.9 find_binding

The find_binding API is used to find UDDI bindingTemplate elements. The find_binding API call returns a bindingDetail that contains zero or more bindingTemplate structures matching the criteria specified in the argument list.

5.1.9.1 Syntax:



Attributes

Name	Use
maxRows	optional
serviceKey	optional
listHead	optional

5.1.9.2 Arguments:

- ***authInfo****:* This optional argument is an element that contains an authentication token. Registries that wish to restrict who can perform an inquiry typically require authInfo for this call.
- ***categoryBag***: This optional argument is a list of category references in the form of keyedReference elements and keyedReferenceGroup structures. When used, the returned bindingDetail for this API will contain elements matching all of the categories passed (logical AND by default). Specifying the appropriate findQualifiers can override this behavior. Matching rules for each can be found in Section 5.1.7 *Matching Rules for keyedReferences and keyedReferenceGroups*.
- ***findQualifiers***: This optional collection of findQualifier elements can be used to alter the default behavior of search functionality. See Section 5.1.4 *Find Qualifiers*, for more information.
- ***find_tModel*****:** This argument provides an alternative or additional way of specifying tModelKeys that are to be used to find the bindingTemplate elements. When specified, the find_tModel argument is treated as an embedded inquiry that is performed prior to searching for bindingTemplate elements. The tModelKeys found are those whose tModels match the criteria contained within the find_tModel element. The tModelKeys found are added to the (possibly empty) collection specified by the tModelBag prior to finding qualified bindingTemplates. Note that the authInfo argument to this embedded find_tModel argument is always ignored. Large result set behavior involving the return of a listDescription does not apply within an embedded argument. If the intermediate result set produced is too large, then the overall query will return E_resultSetTooLarge with an indication that the embedded query returned too many results. If an E_unsupported error occurs as part of processing this embedded argument, it is propagated up to the containing (calling) API.
- ***listHead***: This optional integer value is used to indicate which item SHOULD be returned as the head of the list. The client may request a subset of the matching data by indicating which item in the resultant set constitutes the beginning of the returned data. The use of the listDescription element is mutually exclusive to the use of the truncated attribute that simply indicates a truncated result list in the Inquiry APIs. See Section 5.1.5 *Use of listDescription*, for a detailed description of the listHead argument.
- ***maxRows***: This optional integer value allows the requesting program to limit the number of results returned. This argument can be used in conjunction with the listHead argument.
- ***serviceKey***: This optional uddi_key is used to specify a particular instance of a businessService element in the registered data. Only bindings in the specific businessService data identified by the serviceKey passed are searched. When it is either omitted or specified as empty (i.e., serviceKey=""), this indicates that all

businessServices are to be searched for bindings that meet the other criteria supplied without regard to the service that provides them, and "projected" services are suppressed.

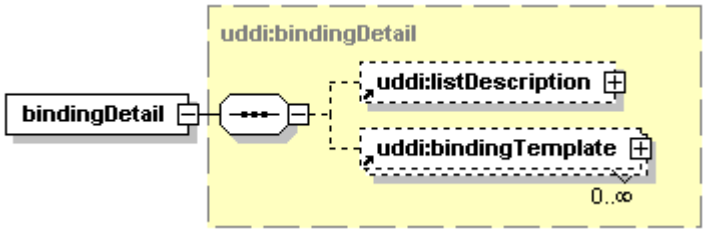
- ***tModelBag*:** This collection of tModelKey elements represent in part or in whole the technical fingerprint of the bindingTemplate structures for which the search is being performed. At least one of either a tModelBag or a find_tModel argument SHOULD be supplied, unless a categoryBag based search is being used.

  If a find_tModel argument is specified (see above), it is treated as an embedded inquiry. The tModelKeys returned as a result of this embedded find_tModel argument are used as if they had been supplied in a tModelBag argument. Changing the order of the keys in the collection or specifying the same tModelKey more than once does not change the behavior of the find.

  By default, only bindingTemplates that have a technical fingerprint containing all of the supplied tModelKeys match (logical AND). Specifying appropriate findQualifiers can override this behavior so that bindingTemplates with a technical fingerprint containing any of the specified tModelKeys are returned (logical OR).

5.1.9.3 Returns:

This API call returns a bindingDetail upon success:



Attributes

Name	Use
truncated	optional

In the event that no matches were located for the specified criteria, the bindingDetail structure returned is empty (i.e., it contains no bindingTemplate data). This signifies a zero match result. If no arguments are passed, a zero-match result set will be returned.

If the number of matches exceeds the value of the maxRows attribute, the result set MAY be truncated. If this occurs, the response contains the attribute "truncated " with the value "true".

As an alternative to the truncated attribute, a registry MAY return a listDescription element. See Section 5.1.5 *Use of listDescription*, for additional information.

5.1.9.4 Caveats:

If an error occurs in processing this API, a dispositionReport element is returned to the caller within a SOAP Fault. In addition to the errors common to all APIs, the following error information is relevant here:

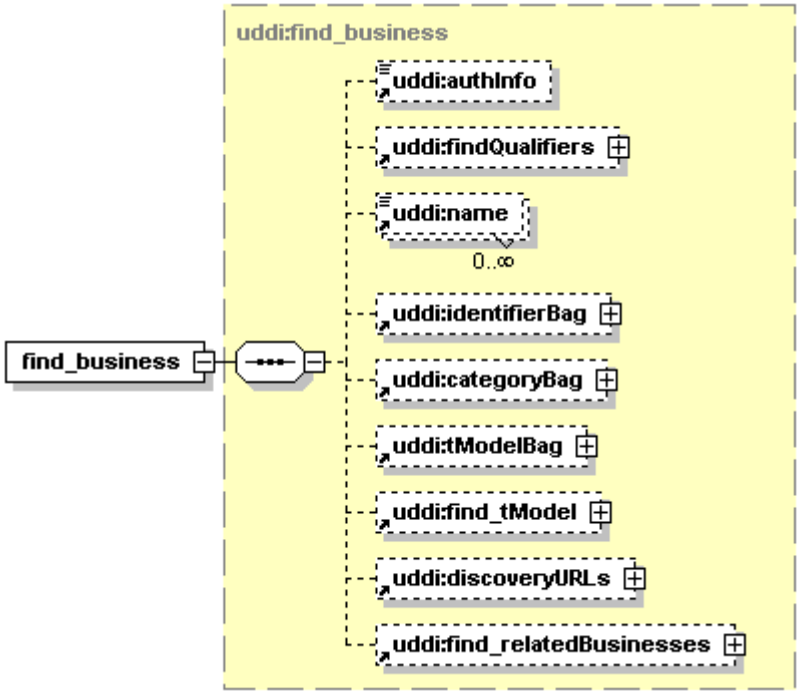
- **E_invalidCombination**: signifies that conflicting findQualifiers have been specified. The error text clearly identifies the findQualifiers that caused the problem.
- **E_invalidKeyPassed**: signifies that the uddi_key value passed did not match with any known serviceKey or tModelKey values. The error structure signifies the condition that occurred and the error text clearly calls out the offending key.

- **E_resultSetTooLarge**: signifies that the particular node queried has deemed that the entire result set is too large to manage. Therefore, the result set is not available. Search criteria must be adjusted to obtain a result.
- **E_unsupported**: signifies that one of the findQualifier values passed was invalid. The invalid qualifier will be indicated clearly in text.

5.1.10 find_business

The find_business API is used to find UDDI businessEntity elements. The find_business API call returns a businessList that matches the criteria specified in the arguments.

5.1.10.1 Syntax:



Attributes

Name	Use
maxRows	optional
listHead	optional

5.1.10.2 Arguments:

- ***authInfo***: This optional argument is an element that contains an authentication token. Registries that wish to restrict who can perform an inquiry in them typically require authInfo for this call.
- ***categoryBag***: This is a list of category references in the form of keyedReference elements and keyedReferenceGroup structures. The returned businessList contains businessInfo elements matching all of the categories passed (logical AND by default). Specifying the appropriate findQualifiers can override this behavior. Matching rules for each can be found in Section 5.1.7 *Matching Rules for keyedReferences and keyedReferenceGroups*.

- ***discoveryURLs***: This is a list of discoveryURL structures to be matched against the discoveryURL data associated with registered businessEntity information. To search for URL without regard to useType attribute values, omit the useType attribute or pass it as an empty attribute. If useType values are included, the match occurs only on registered information that matches both the useType and URL value. The returned businessList contains businessInfo structures matching any of the URL's passed (logical OR).
- ***identifierBag*****:** This is a list of business identifier references in the form of keyedReference elements. The returned businessList contains businessInfo structures matching any of the identifiers passed (logical OR by default). Specifying the appropriate findQualifiers can override this behavior. Matching rules can be found in Section 5.1.7 *Matching Rules for keyedReferences and keyedReferenceGroups*.
- ***findQualifiers***: This collection of findQualifier elements can be used to alter the default behavior of search functionality. See the Section 5.1.4 *Find Qualifiers*, for more information.
- ***find_relatedBusinesses***: This argument is an embedded inquiry and limits the search results to those businesses that are related to a specified business in a specified way. The result is comprised of an intersection of the businesses located with this embedded inquiry and the businesses discovered using the remaining inquiry criteria. The standard syntax and arguments for find_relatedBusinesses apply here. Note that the authInfo argument to this embedded find_relatedBusinesses argument is always ignored. Large result set behavior involving the return of a listDescription does not apply within an embedded argument. If the intermediate result set produced is too large, then the overall query will return E_resultSetTooLarge with an indication that the embedded query returned too many results. If an E_unsupported error occurs as part of processing this embedded argument, it is propagated up to the containing (calling) API. See Section 5.1.11 *find_relatedBusinesses*, for further information.
- ***find_tModel*****:** This argument provides an alternative or additional way of specifying tModelKeys that are used to find businesses which have service bindings with specific technical fingerprints as described above for the tModelBag element. When specified, the find_tModel argument is treated as an embedded inquiry that is performed prior to searching for businesses. The tModelKeys found are those whose tModels match the criteria contained within the find_tModel element. The tModelKeys found are added to the (possibly empty) collection specified by the tModelBag prior to finding qualified businesses. Note that the authInfo argument to this embedded find_tModel argument is always ignored. Large result set behavior involving the return of a listDescription does not apply within an embedded argument. If the intermediate result set produced is too large, then the overall query will return E_resultSetTooLarge with an indication that the embedded query returned too many results. If an E_unsupported error occurs as part of processing this embedded argument, it is propagated up to the containing (calling) API.
- ***listHead***: This optional integer value is used to indicate which item SHOULD be returned as the head of the list. The client may request a subset of the matching data by indicating which item in the resultant set constitutes the beginning of the returned data. The use of the listDescription element is mutually exclusive to the use of the truncated attribute that simply indicates a truncated result list in the Inquiry APIs. See Section 5.1.5 *Use of listDescription*, for a detailed description of the listHead argument.
- ***maxRows***: This optional integer value allows the requesting program to limit the number of results returned. This argument can be used in conjunction with the listHead argument.

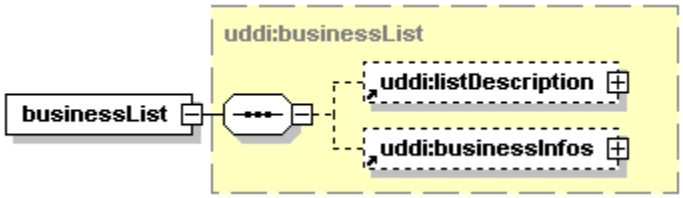
- ***name*:** This optional collection of string values represents one or more names potentially qualified with xml:lang attributes. Since "exactMatch" is the default behavior, the value supplied for the name argument must be an exact match. If the "approximateMatch" findQualifier is used together with an appropriate wildcard character in the name, then any businessEntity matching this name with wildcards and the other criteria will be referenced in the results. For more on wildcard matching, see Section 5.1.6 *About Wildcards*. The businessList returned contains businessInfo structures for businesses whose name matches the value(s) passed (lexical-order match – i.e., leftmost in left-to-right languages). If multiple name values are passed, the match occurs on a logical OR basis. Each name MAY be marked with an xml:lang adornment. If a language markup is specified, the search results report a match only on those entries that match both the name value and language criteria. The match on language is a leftmost case-insensitive comparison of the characters supplied. This allows one to find all businesses whose name begins with an "A" and are expressed in any dialect of French, for example. Values which can be passed in the language criteria adornment MUST obey the rules governing the xml:lang data type as defined in Section 3.3.2.3 *name*.
- ***tModelBag*:** Every Web service instance exposed by a registered businessEntity is represented in UDDI by a bindingTemplate contained within the businessEntity. Each bindingTemplate contains a collection of tModel references called its "technical fingerprint" that specifies its type. The tModelBag argument is a collection of tModelKey elements specifying that the search results are to be limited to businesses that expose Web services with technical fingerprints that match.

  If a find_tModel argument is specified (see above), it is treated as an embedded inquiry. The tModelKeys returned as a result of this embedded find_tModel argument are used as if they had been supplied in a tModelBag argument. Changing the order of the keys in the collection or specifying the same tModelKey more than once does not change the behavior of the find.

  By default, only bindingTemplates that contain all of the tModelKeys in the technical fingerprint match (logical AND). Specifying appropriate findQualifiers can override this behavior so that bindingTemplates containing any of the specified tModelKeys match (logical OR).

5.1.10.3 Returns:

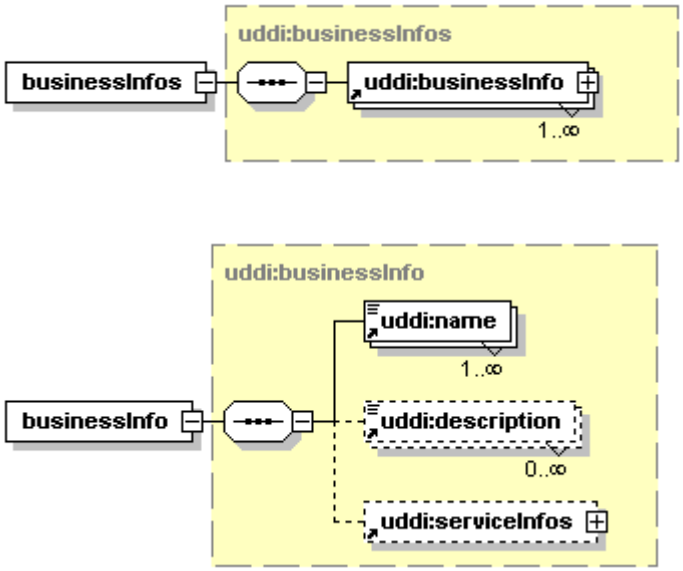
This API call returns a businessList on success. This structure contains information about each matching business, including summaries of its businessServices:



Attributes

Name	Use
truncated	optional

The businessList's businessInfos structure and its businessInfo structures contain:



Attributes

Name	Use
businessKey	required

If a tModelBag or find_tModel was used in the search, the resulting serviceInfos structure reflects data only for the businessServices that actually contained a matching bindingTemplate. For more information on serviceInfos, see Section 5.1.12.3 *[find_service] Returns*.

Projected services are treated exactly the same as services that are naturally a part of businessEntities unless the suppressProjectedServices findQualifier is specified, in which case they are omitted from the serviceInfos structure returned and are not considered when determining which businesses match the inquiry criteria. In the event that no matches are found for the specified criteria, a businessList structure containing no businessInfos structure is returned.

In the event that no matches were located for the specified criteria, the businessList structure returned is empty (i.e., it contains no businessInfos data). This signifies a zero match result. If no arguments are passed, a zero-match result set will be returned.

In the event of a large number of matches, (as determined by the UDDI node), or if the number of matches exceeds the value of the maxRows attribute, the UDDI node MAY truncate the result set. If this occurs, the businessList will contain the attribute "truncated" with the value "true".

Second level elements (serviceInfos) within the returned businessList will be sorted in the order in which they were saved.

As an alternative to the truncated attribute, a registry MAY return a listDescription element. See Section 5.1.5 *Use of listDescription*, for additional information.

5.1.10.4 Caveats:

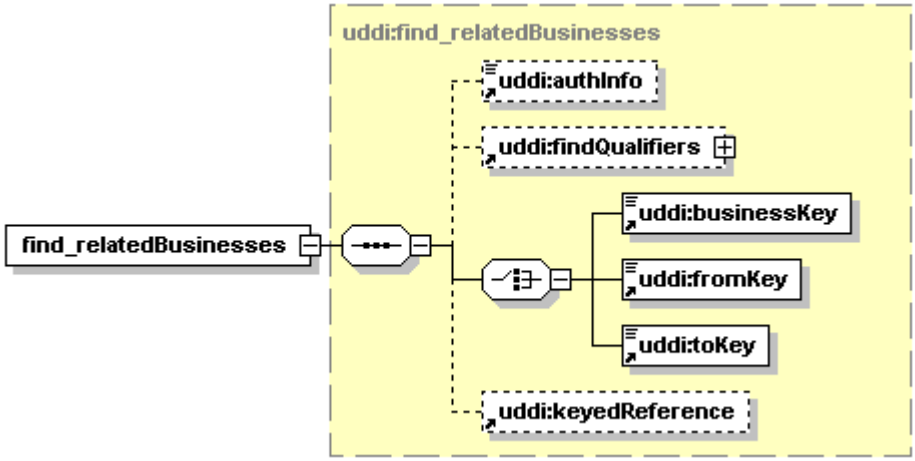
If any error occurs in processing this API call, a dispositionReport structure is returned to the caller in a SOAP Fault. In addition to the errors common to all APIs, the following error information is relevant here:

- **E_invalidCombination**: signifies that conflicting findQualifiers have been specified. The error text clearly identifies the findQualifiers that caused the problem.
- **E_unsupported**: signifies that one of the findQualifier values passed was invalid. The findQualifier value that was not recognized will be clearly indicated in the error text.
- **E_invalidKeyPassed**: signifies that a *uddi_key* or tModelKey value passed did not match with any known businessKey key or tModelKey values. The error structure signifies the condition that occurred and the error text clearly calls out the offending key.
- **E_resultSetTooLarge**: signifies that the node deems that a result set from an inquiry is too large and does not honor requests to obtain the results for this inquiry, even using subsets. The inquiry that triggered this error SHOULD be refined and re-issued.

5.1.11 find_relatedBusinesses

The find_relatedBusinesses API is used to find businessEntity elements, which have a completed relationship with the specified businessEntity that matches the criteria supplied. The find_relatedBusinesses API call returns a relatedBusinessesList structure containing results that match the criteria specified in the arguments. For additional information on the use of find_relatedBusinesses, refer to Appendix A: *Relationships and Publisher Assertions,* for more information on business relationships.

5.1.11.1 Syntax:



Attributes

Name	Use
maxRows	optional
listHead	optional

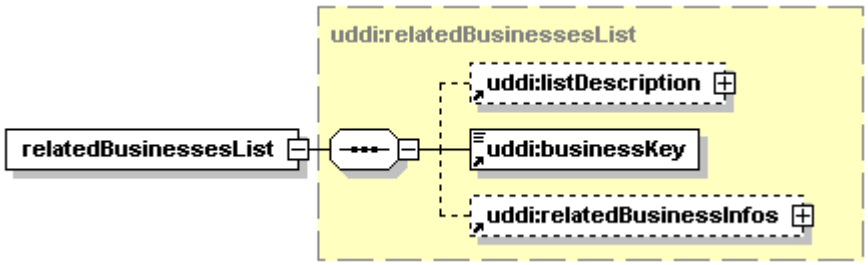
5.1.11.2 Arguments:

- ***authInfo***: This optional argument is an element that contains an authentication token. Registries that wish to restrict who can perform an inquiry in them typically require authInfo for this call.

- ***businessKey***: This uddi_key is used to specify a particular businessEntity instance to use as the focal point of the search. It MUST NOT be used in conjunction with the fromKey or toKey arguments. It MUST refer to the businessKey of an existing businessEntity in the registry. The result set reports businesses that are related in some way to the businessEntity whose key is specified.
- ***findQualifiers***: This collection of findQualifier elements can be used to alter the default behavior of search functionality. See Section 5.1.4 *Find Qualifiers*, for more information.
- ***fromKey***: This uddi_key is used to specify a particular businessEntity instance which corresponds to the fromKey of a completed businessRelationship, for use as the focal point of the search. It MUST NOT be used in conjunction with the businessKey or toKey arguments. The result set reports businesses for which a relationship whose fromKey matches the key specified exists.
- ***keyedReference***: This is a single, optional keyedReference element that is used to specify a relationship type, such that only businesses that are related to the focal point in a specific way are included in the results. Specifying a keyedReference only affects whether a business is selected for inclusion in the results. If a business is selected, the results include the full set of completed relationships between it and the focal point. See Appendix A: *Relationships and Publisher Assertions,* for more information on specifying relationships. Matching rules for the use of keyedReferences are described in Section 5.1.7 *Matching Rules for keyedReferences and keyedReferenceGroups*, with the exception that keyNames MUST also match if they are non-empty in the search criteria for this API. Omitted keyNames are treated as empty keyNames.
- ***listHead***: This optional integer value is used to indicate which item SHOULD be returned as the head of the list. The client may request a subset of the matching data by indicating which item in the resultant set constitutes the beginning of the returned data. The use of the listDescription element is mutually exclusive to the use of the truncated attribute that simply indicates a truncated result list in the Inquiry APIs. See Section 5.1.5 *Use of listDescription*, for a detailed description of the listHead argument.
- ***maxRows***: This optional integer value allows the requesting program to limit the number of results returned. This argument can be used in conjunction with the listHead argument.
- ***toKey***: This uddi_key is used to specify a particular businessEntity instance which corresponds to the toKey of an existing businessRelationship, for use as the focal point of the search. It MUST NOT be used in conjunction with the businessKey or fromKey arguments. The result set reports businesses for which a relationship whose toKey matches the key specified exists.

5.1.11.3 Returns:

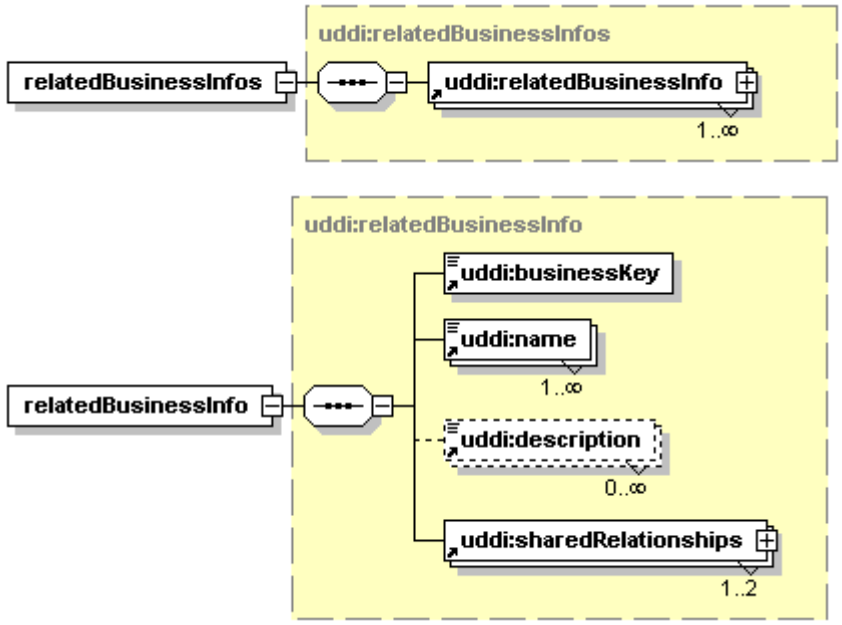
This API call returns a relatedBusinessesList on success:



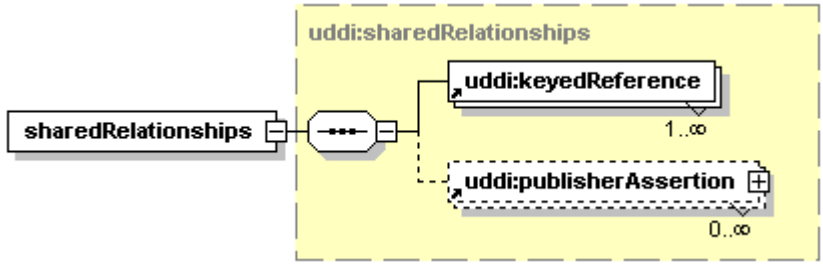
Attributes

Name	Use
truncated	optional

Here, the businessKey returned with relatedBusinessesList is the same one provided with the find_relatedBusinesses API call. The relatedBusinessInfos structure and the relatedBusinessInfo structures it contains each have this syntax:



The businessKey returned in each relatedBusinessInfo structure pertains to a business, which matched the search criteria supplied in the find_relatedBusinesses API call. The sharedRelationships structures then have this syntax:



Attributes

Name	Use
direction	required

The value of the direction attribute is determined based on the focal business provided in the query. The focal business is the represented by the key provided as a parameter to the find_relatedBusinesses, specified in the "fromKey", "toKey", or "businessKey" argument.

Only the publisher assertions from completed relationships including the focal business contribute to the results of find_relatedBusinesses. If the focal business is specified in the query as the "fromKey" then only those assertions where the focal business is the "fromKey" will contribute to the results. Conversely, if the focal business is specified in the query as the "toKey" then only those assertions where the focal business is the "toKey" will be contribute to the results. And if the focal business is specified as the "businessKey" then assertions where the focal business is either the "toKey" and "fromKey" will contribute to the results.

The direction attribute is either expressed as "fromKey" or "toKey" depending on the relationship of the business to those returned by the call.

If the focal business is specified as the fromKey in the find_relatedBusinesses query, the sharedRelationships elements returned will have the direction attribute of fromKey.

If the focal business is specified as the toKey in the find_relatedBusinesses query, the sharedRelationships elements returned will have the direction attribute of toKey.

If the focal business is specified as the businessKey in the find_relatedBusinesses query then there may be one or two sharedRelationships, one marked with the fromKey and one marked with the toKey. Publisher assertions specifying the focal business as the "fromKey" will contribute to the sharedRelationships element with a direction of "fromKey". Correspondingly, publisher assertions specifying the focal business as the "toKey" will contribute to the sharedRelationships element with a direction of "toKey".

The example below depicts that Matt's Garage is related to the focal business (i.e. whose business key is uddi:ws-o-rama-cars.com:be47 and which appeared in the find_relatedBusinesses) by exactly one relationship -- the "subsidiary" parent-child relationship -- and that Matt's Garage is a subsidiary of the focal business. In such cases, the direction attribute is set to "fromKey".

Given the completed relationship resulting from the following publisher assertion:

```
<publisherAssertion>
  <!-- Specify ws-o-rama-cars.com:be47 businessKey as fromKey-->
  <fromKey>
    uddi:ws-o-rama-cars.com:be47
  </fromKey>
  <!-- Specify ws-o-rama-cars.com:mattsgarage:be3's businessKey as toKey-->
  <toKey>
    uddi:ws-o-rama-cars.com:mattsgarage:be3
  </toKey>
  <!--Specify a subsidiary relationship using uddi-org:relationships -->
  <keyedReference keyName="Subsidiary"
    keyValue="parent-child"
    tModelKey="uddi:uddi.org:relationships"/>
</publisherAssertion>
```

and the following find_relatedBusinesses query:

```
<find_relatedBusinesses xmlns="urn:uddi-org:api_v3">
<businessKey>uddi:ws-o-rama-cars.com:be47</businessKey>
</find_relatedBusinesses>
```

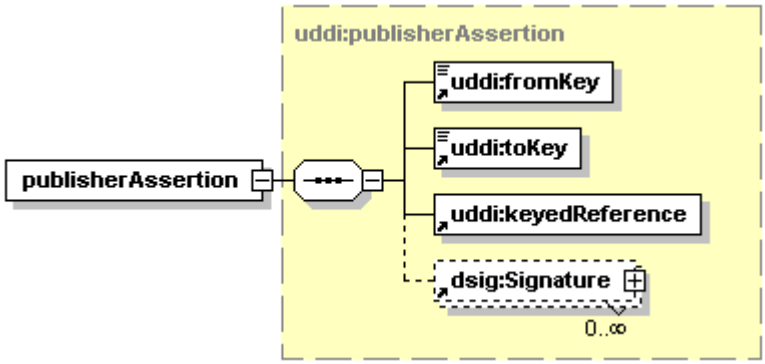
the following relatedBusinessList will be returned:

```
<relatedBusinessesList operator="uddi.someoperator" truncated="false"
xmlns="urn:uddi-org:api_v3">
<businessKey>uddi:ws-o-rama-cars.com:be47</businessKey>
   <relatedBusinessInfos>
        <relatedBusinessInfo>
            <businessKey>uddi:ws-o-rama-cars.com:mattsgarage:be3</businessKey>
            <name>Matt's Garage</name>
            <description>Car services in …</description>
            <sharedRelationships direction="fromKey">
                <keyedReference tModelKey="uddi:uddi.org:relationships"
```

```
                keyName="Subsidiary"
                keyValue="parent-child">
            …]
    […]
```

In a relatedBusinessInfo with two sharedRelationships elements the sharedRelationships element with direction attribute value of "fromKey" precedes the one with the value of "toKey". The keyedReference elements in the sharedRelationships element will be sorted by last date change of the corresponding publisher assertion in ascending order.

A publisherAssertion structure is only returned if it contains a signature and it has the following syntax:



In the event that no matches were located for the specified criteria, the relatedBusinessesList structure returned does not contain a relatedBusinessInfos element. This signifies zero matches.

In the event of a large number of matches (as determined by the node), or if the number of matches exceeds the value of the maxRows attribute, the node MAY truncate the result set. When this occurs, the relatedBusinessesList contains the attribute "truncated" with the value of this attribute set to "true".

As an alternative to use of the truncated attribute, a registry MAY return a listDescription element. See Section 5.1.5 *Use of listDescription*, for additional information.

5.1.11.4 Caveats:

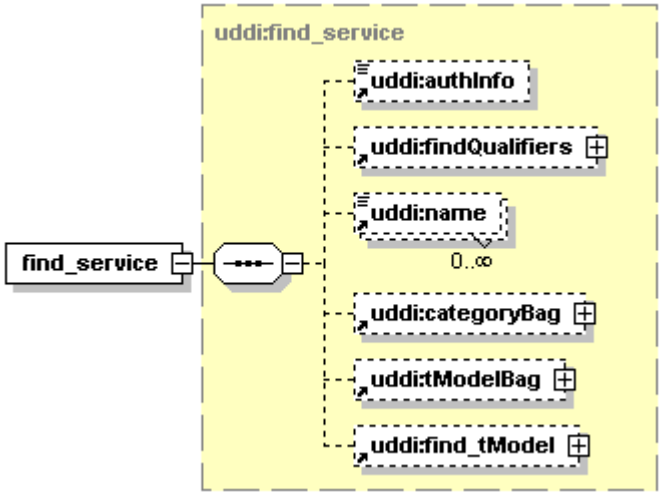
If any error occurs in processing this API call, a dispositionReport structure is returned to the caller in a SOAP Fault. In addition to the errors common to all APIs, the following error information is relevant here:

- **E_invalidCombination**: signifies that conflicting findQualifiers have been specified. The error text clearly identifies the findQualifiers that caused the problem.
- **E_invalidKeyPassed**: signifies that an uddi_key or tModelKey value passed did not match with any known businessKey key or tModelKey values. The error structure signifies that the condition occurred and the error text clearly calls out the offending key.
- **E_unsupported**: signifies that one of the findQualifier values passed was invalid. The findQualifier value that was not recognized will be clearly indicated in the error text.
- **E_resultSetTooLarge**: signifies that the node deems that a result set from an inquiry is too large and does not honor requests to obtain the results for this inquiry, even using subsets. The inquiry that triggered this error SHOULD be refined and re-issued.

5.1.12 find_service

The find_service API is used to find UDDI businessService elements. The find_service API call returns a serviceList structure that matches the conditions specified in the arguments.

5.1.12.1 Syntax:



Attributes

Name	Use
maxRows	optional
businessKey	optional
listHead	optional

5.1.12.2 Arguments:

- ***authInfo***: This optional argument is an element that contains an authentication token. Registries that wish to restrict who can perform an inquiry in them typically require authInfo for this call.
- ***businessKey***: This optional uddi_key is used to specify a particular businessEntity instance to search. When supplied, this argument is used to specify an existing businessEntity within which services should be found. Projected services are included unless the "suppressProjectedServices" findQualifier is used. If businessKey it is either omitted or specified as empty (i.e., businessKey=""), this indicates that all businessEntities are to be searched for services that meet the other criteria supplied without regard to the business that provides them and service projections does not apply.
- ***categoryBag***: This is a list of category references. The returned serviceList contains serviceInfo structures matching all of the categories passed (logical AND by default). Specifying the appropriate findQualifiers can override this behavior. Matching rules for the use of keyedReferences and keyedReferenceGroups are described in Section 5.1.7 *Matching Rules for keyedReferences and keyedReferenceGroups*.

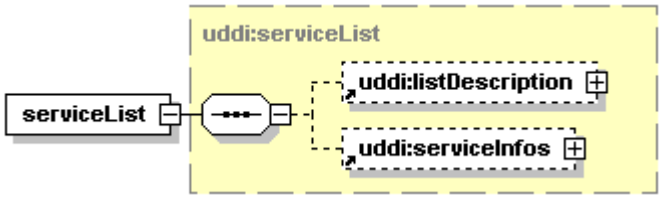
- ***findQualifiers***: This optional collection of findQualifier elements can be used to alter the default behavior of search functionality. See Section 5.1.4 *Find Qualifiers*, for more information.
- ***find_tModel*:** This argument provides an alternative or additional way of specifying tModelKeys that are used to find services which have service bindings with specific technical fingerprints, as described above for the tModelBag element. When specified, the find_tModel argument is treated as an embedded inquiry that is performed prior to searching for services. The tModelKeys found are those whose tModels match the criteria contained within the find_tModel element. The tModelKeys found are added to the (possibly empty) collection specified by the tModelBag prior to finding qualified services. Note that the authInfo argument to this embedded find_tModel argument is always ignored. Large result set behavior involving the return of a listDescription does not apply within an embedded argument. If an E_unsupported error occurs as part of processing this embedded argument, it is propagated up to the containing (calling) API.
- ***listHead***: This optional integer value is used to indicate which item SHOULD be returned as the head of the list. The client may request a subset of the matching data by indicating which item in the resultant set constitutes the beginning of the returned data. The use of the listDescription element is mutually exclusive to the use of the truncated attribute that simply indicates a truncated result list in the Inquiry APIs. See Section 5.1.5 *Use of listDescription*, for a detailed description of the listHead argument.
- ***maxRows***: This optional integer value allows the requesting program to limit the number of results returned. This argument can be used in conjunction with the listHead argument.
- ***name*:** This optional collection of string values represents one or more names potentially qualified with xml:lang attributes. Since "exactMatch" is the default behavior, the value supplied for the name argument must be an exact match. If the "approximateMatch" findQualifier is used together with an appropriate wildcard character in the name, then any businessService data contained in the specified businessEntity (or across all businesses if the businessKey is omitted or specified as empty) with matching name value will be returned. Matching occurs using wildcard matching rules. See Section 5.1.6 *About Wildcards*. If multiple name values are passed, the match occurs on a logical OR basis within any names supplied. Each name MAY be marked with an xml:lang adornment. If a language markup is specified, the search results report a match only on those entries that match both the name value and language criteria. The match on language is a leftmost case-insensitive comparison of the characters supplied. This allows one to find all services whose name begins with an "A" and are expressed in any dialect of French, for example. Values which can be passed in the language criteria adornment MUST obey the rules governing the xml:lang data type as defined in Section 3.3.2.3 *name*.
- ***tModelBag***: Every Web service instance is represented in UDDI by a bindingTemplate contained within some businessService. A bindingTemplate contains a collection of tModel references called its "technical fingerprint" that specifies its type. The tModelBag argument is a collection of tModelKey values specifying that the search results are to be limited to businessServices containing bindingTemplates with technical fingerprints that match.

  If a find_tModel argument is specified (see below), it is treated as an embedded inquiry. The tModelKeys returned as a result of this embedded find_tModel argument are used as if they had been supplied in a tModelBag argument. Changing the order of the keys in the collection or specifying the same tModelKey more than once does not change the behavior of the find.

By default, only bindingTemplates that contain all of the tModelKeys in the technical fingerprint match (logical AND). Specifying appropriate findQualifiers can override this behavior so that bindingTemplates containing any of the specified tModelKeys match (logical OR).

5.1.12.3 Returns:

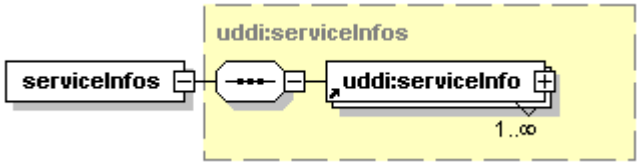
This API call returns a serviceList on success:

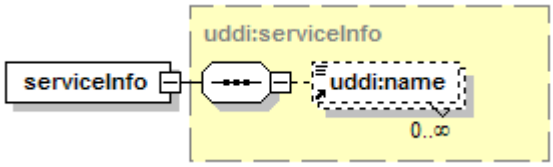


Attributes

Name	Use
truncated	optional

The serviceInfos and contained serviceInfo structures have the syntax:





Attributes

Name	Use
serviceKey	required
businessKey	required

In the event that no matches were located for the specified criteria, the serviceList structure returned does not contain a serviceInfos element. This signifies zero matches. If no search arguments (including businessKey) are passed, a zero-match result set is returned. If only the businessKey search argument is passed, the entire set of services for the business is returned. The named arguments are all optional, and with the exception of name, may appear at most once. When more than one distinct named argument is passed, matching services are those which match on all of the criteria.

When a businessKey is supplied, the resulting serviceList contains only services that are associated with the designated business. Service projections are included in this list unless explicitly excluded using the *suppressProjectedServices* find Qualifier. When the businessKey is omitted or specified as empty (i.e., businessKey=""), all services that meet the other criteria are returned in the serviceList, without regard to the business which own them. Service

projections are not returned when a businessKey is omitted or left empty because they are exact duplicates of the services being projected upon.

Since a serviceInfo structure can represent a projection to a deleted businessService, the name element within the serviceInfo structure is optional (see Section 5.2.16 *save_business* on deleting projected services).

In the event of a large number of matches (as determined by the node), or if the number of matches exceeds the value of the maxRows attribute, the result set MAY be truncated. When this occurs, the serviceList contains the attribute "truncated" with the value of this attribute set to "true".

As an alternative to the truncated attribute, a registry MAY return a listDescription element. See Section 5.1.5 *Use of listDescription* for additional information.

5.1.12.4 Caveats:

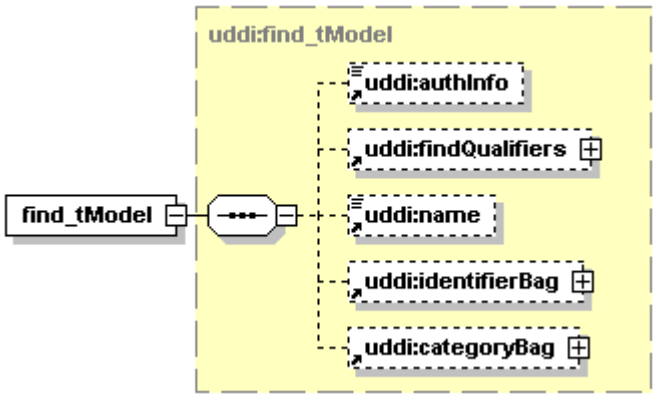
If any error occurs in processing this API call, a dispositionReport structure is returned to the caller in a SOAP Fault. In addition to the errors common to all APIs, the following error information is relevant here:

- **E_invalidCombination**: signifies that conflicting findQualifiers have been specified. The error text clearly identifies the findQualifiers that caused the problem.
- **E_invalidKeyPassed**: signifies that the uddi_key value passed did not match with any known businessKey key or tModelKey values. The error structure signifies the condition that occurred and the error text clearly calls out the offending key.
- **E_unsupported**: signifies that one of the findQualifier values passed was invalid. The findQualifier value that was not recognized will be clearly indicated in the error text.
- **E_resultSetTooLarge**: signifies that the node deems that a result set from an inquiry is too large and does not honor requests to obtain the results for this inquiry, even using subsets. The inquiry that triggered this error SHOULD be refined and re-issued.

5.1.13 find_tModel

The find_tModel API is used to find UDDI tModel elements. The find_tModel API call returns a list of tModel entries that match a set of specific criteria. The response consists of summary information about registered tModel data returned in a tModelList structure.

5.1.13.1 Syntax:



Attributes

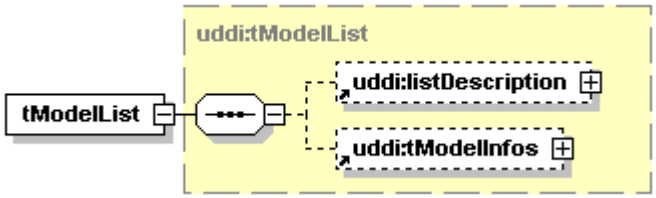
Name	Use
maxRows	optional
listHead	optional

5.1.13.2 Arguments:

- ***authInfo***: This optional argument is an element that contains an authentication token. Registries that wish to restrict who can perform an inquiry in them typically require authInfo for this call.
- ***categoryBag***: This is a list of category references. The returned tModelList contains tModelInfo elements whose associated tModels match all of the categories passed (logical AND by default). Specifying the appropriate findQualifiers can override this behavior. Matching rules for the use of keyedReferences and keyedReferenceGroups are described in Section 5.1.7 *Matching Rules for keyedReferences and keyedReferenceGroups*.
- ***findQualifiers***: This collection of findQualifier elements is used to alter the default behavior of search functionality. See Section 5.1.4 *Find Qualifiers* for more information.
- ***identifierBag*** This is a list of identifier references in the form of keyedReference elements. The returned tModelList contains tModelInfo elements whose associated tModels match any of the identifiers passed (logical OR by default). Specifying the appropriate findQualifiers can override this behavior. Matching rules are described in Section 5.1.7 *Matching Rules for keyedReferences and keyedReferenceGroups*.
- ***listHead***: This optional integer value is used to indicate which item SHOULD be returned as the head of the list. The client may request a subset of the matching data by indicating which item in the resultant set constitutes the beginning of the returned data. The use of the listDescription element is mutually exclusive to the use of the truncated attribute that simply indicates a truncated result list in the Inquiry APIs. See Section 5.1.5 *Use of listDescription*, for a detailed description of the listHead argument.
- ***maxRows***: This optional integer value allows the requesting program to limit the number of results returned. This argument can be used in conjunction with the listHead argument.
- ***name*****:** This string value represents the name of the tModel elements to be found. Since tModel data only has a single name, only a single name may be passed. The argument must match exactly since "exactMatch" is the default behavior, but if the "approximateMatch" findQualifier is used together with the appropriate wildcard character, then matching is done according to wildcard rules. See Section 5.1.6 *About Wildcards* for additional information. The name MAY be marked with an xml:lang adornment. If a language markup is specified, the search results report a match only on those entries that match both the name value and language criteria. The match on language is a leftmost case-insensitive comparison of the characters supplied. This allows one to find all tModels whose name begins with an "A" and are expressed in any dialect of French, for example. Values which can be passed in the language criteria adornment MUST obey the rules governing the xml:lang data type as defined in Section 3.3.2.3 *name*.

5.1.13.3 Returns:

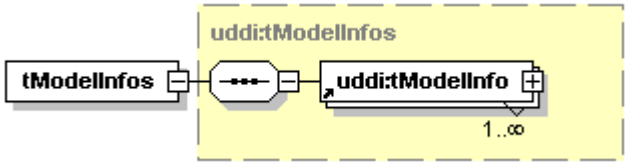
This API call returns a tModelList structure on success:

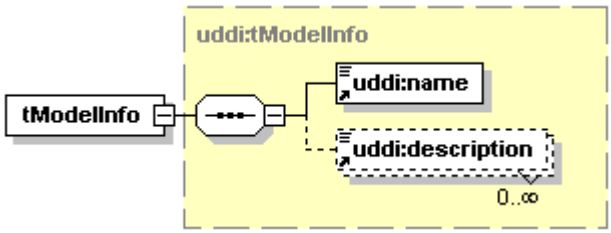


Attributes

Name	Use
truncated	optional

The tModelInfos and contained tModelInfo structures have the syntax:





Attributes

Name	Use
tModelKey	required

In the event that no matches were located for the specified criteria, the tModelList returned will not contain a tModelInfos element. This signifies zero matches. If no arguments are passed, a zero-match result is returned.

In the event of a large number of matches (as determined by the node), or if the number of matches exceeds the value of the maxRows attribute, the result set MAY be truncated. When this occurs, the tModelList contains the attribute "truncated" with the value "true".

As an alternative to the truncated attribute, a registry MAY return a listDescription element. See Section 5.1.5 *Use of listDescription* for additional information.

5.1.13.4 Caveats:

If any error occurs in processing this API call, a dispositionReport element is returned to the caller within a SOAP Fault. In addition to the errors common to all APIs, the following error information is relevant here:

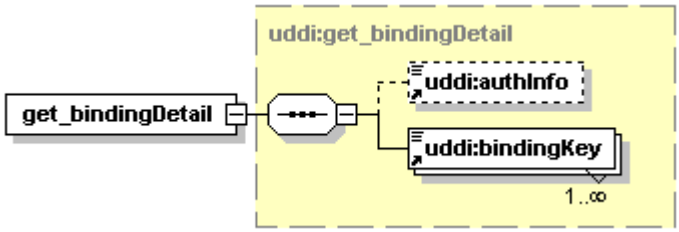
- **E_invalidCombination**: signifies that conflicting findQualifiers have been specified. The error text clearly identifies the findQualifiers that caused the problem.

- **E_invalidKeyPassed**: signifies that the uddi_key value passed did not match with any known tModelKey values. The error structure signifies the condition that occurred and the error text clearly calls out the offending key.
- **E_unsupported**: signifies that one of the findQualifier values passed was invalid. The invalid qualifier is clearly indicated in the error text.
- **E_resultSetTooLarge**: signifies that the node deems that a result set from an inquiry is too large and does not honor requests to obtain the results for this inquiry, even using subsets. The inquiry that triggered this error SHOULD be refined and re-issued.

5.1.14 get_bindingDetail

The get_bindingDetail API call returns the bindingTemplate structure corresponding to each of the bindingKey values specified.

5.1.14.1 Syntax:



5.1.14.2 Arguments:

- ***authInfo***: This optional argument is an element that contains an authentication token. Registries that wish to restrict who can perform an inquiry in them typically require authInfo for this call.
- ***bindingKey***: One or more uddi_key values that represent the UDDI assigned keys for specific instances of registered bindingTemplate data.

5.1.14.3 Returns:

This API call returns a bindingDetail on successful match of the specified bindingKey values. See Section 5.1.9.3 *[find_binding] Returns* for information on this structure. If multiple bindingKey values were passed, the results are returned in the same order as the keys passed.

If a large number of keys are passed, the node MAY truncate the result set. When this occurs, the bindingDetail result contains the attribute "truncated" with the value "true".

A node MUST NOT return a listDescription element as part of the bindingDetail in response to this API call.

5.1.14.4 Caveats:

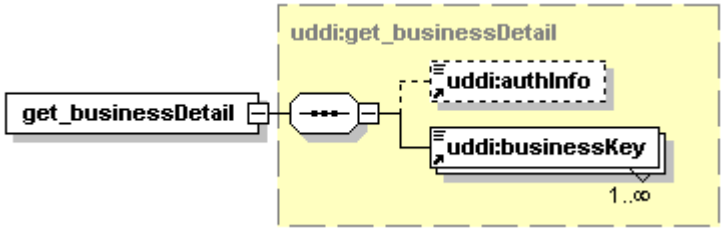
If any error occurs in processing this API call, a dispositionReport structure is returned to the caller in a SOAP Fault. In addition to the errors common to all APIs, the following error information is relevant here:

- **E_invalidKeyPassed**: signifies that one of the uddi_key values passed did not match with any known bindingKey key values. No partial results are returned – if any bindingKey values passed are not valid bindingKey values, this error is returned. The error text clearly calls out the offending key.

5.1.15 get_businessDetail

The get_businessDetail API call returns the businessEntity structure corresponding to each of the businessKey values specified.

5.1.15.1 Syntax:

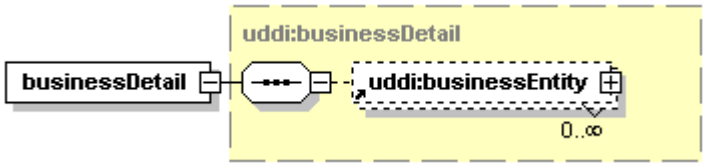


5.1.15.2 Arguments:

- ***authInfo****:* This optional argument is an element that contains an authentication token. Registries that wish to restrict who can perform an inquiry in them typically require authInfo for this call.
- ***businessKey***: One or more uddi_key values that represent specific instances of known businessEntity data.

5.1.15.3 Returns:

This API call returns a businessDetail on successful match of the specified businessKey values:



Attributes

Name	Use
truncated	optional

If multiple businessKey values were passed, the results MUST be returned in the same order as the keys passed.

If a large number of keys are passed, a node MAY truncate the result set. When this occurs, the businessDetail response contains the attribute "truncated " with the value "true".

businessEntity elements retrieved with get_businessDetail can contain service projections. Such projected services appear in full in the businessEntity in which they occur. Projected services can be distinguished from the services that are naturally contained in the businessEntity in which they appear by their businessKey. Services naturally contained in the businessEntity have the businessKey of the businessEntity in which they appear. Projected services have the businessKey of the businessEntity of which they are a natural part.

5.1.15.4 Caveats:

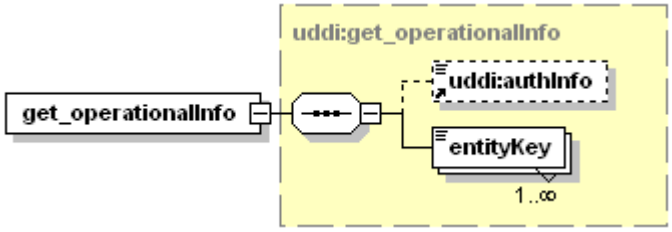
If any error occurs in processing this API call, a dispositionReport element is returned to the caller within a SOAP Fault. In addition to the errors common to all APIs, the following error information is relevant here:

- **E_invalidKeyPassed**: signifies that one of the uddi_key values passed did not match with any known businessKey values. No partial results are returned – if any businessKey values passed are not valid, this error is returned. The error text clearly calls out the offending key.

5.1.16 get_operationalInfo

The get_operationalInfo API call is used to retrieve entity level operational information (such as the date and time that the data structure was created and last modified, the identifier of the UDDI node at which the entity was published and the identity of the publisher) pertaining to one or more entities. The get_operationalInfo API call returns an operationalInfos structure corresponding to each of the entityKey values specified.

5.1.16.1 Syntax:



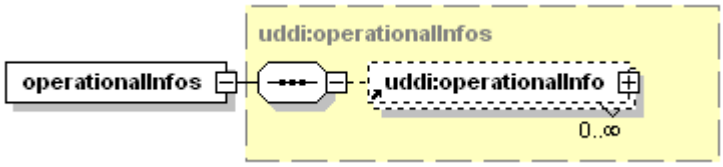
5.1.16.2 Arguments:

- ***authInfo***: This optional argument is an element that contains an authentication token. Registries that wish to restrict who can perform an inquiry in them typically require authInfo for this call.
- ***entityKey***: One or more uddi_key values that represent businessEntity, businessService, bindingTemplate or tModelKeys.

5.1.16.3 Returns:

This API returns an operationalInfos structure that contains an operationalInfo element for each entity requested by the inquirer.

The operationalInfos structure has the form:



Attributes

Name	Use
truncated	optional

For information on the operationalInfo structure, see Section 3.8, *operationalInfo Structure*.

5.1.16.4 Caveats:

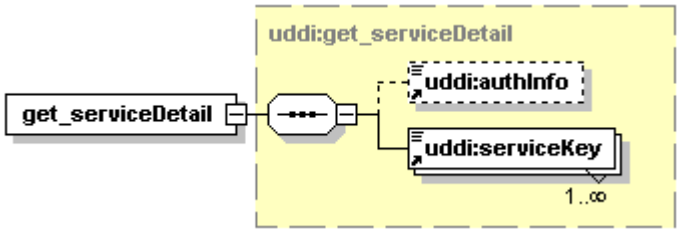
If any error occurs in processing this API call, a dispositionReport element is returned to the caller within a SOAP Fault. In addition to the errors common to all APIs, the following error information is relevant here:

- **E_invalidKeyPassed**: signifies that one of the uddi_key values passed did not match with any known entityKey values. No partial results are returned – if any entityKey values passed are not valid, this error is returned. The error text clearly calls out the offending key(s).

5.1.17 get_serviceDetail

The get_serviceDetail API call returns the businessService structure corresponding to each of the serviceKey values specified.

5.1.17.1 Syntax:

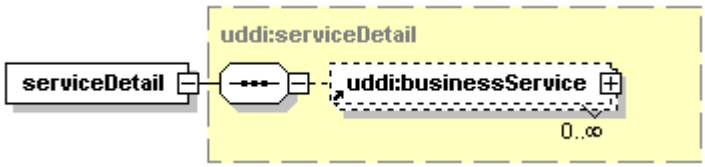


5.1.17.2 Arguments:

- ***authInfo***: This optional argument is an element that contains an authentication token. Registries that wish to restrict who can perform an inquiry in them typically require authInfo for this call.
- ***serviceKey***: One or more uddi_key values that represent UDDI assigned serviceKey values of specific instances of known businessService data.

5.1.17.3 Returns:

This API call returns a serviceDetail on successful match of the specified serviceKey values.



Attributes

Name	Use
truncated	optional

If multiple serviceKey values were passed, the results will be returned in the same order as the keys passed.

If a large number of keys are passed, a node MAY truncate the result set. When this occurs, the response contains the attribute "truncated" with the value "true".

5.1.17.4 Caveats:

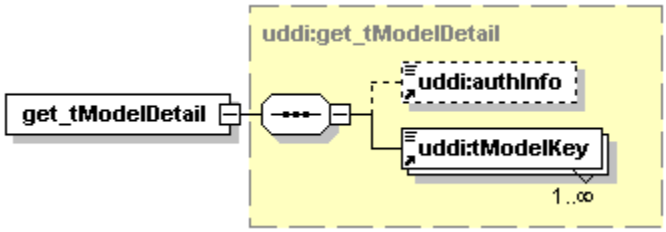
If any error occurs in processing this API call, a dispositionReport element is returned to the caller within a SOAP Fault. In addition to the errors common to all APIs, the following error information is relevant here:

- **E_invalidKeyPassed**: signifies that one of the uddi_key values passed did not match with any known serviceKey values. No partial results are returned – if any serviceKey values passed are not valid, this error is returned. The error text clearly calls out the offending key.

5.1.18 get_tModelDetail

The get_tModelDetail API call returns the tModel structure, corresponding to each of the tModelKey values specified.

5.1.18.1 Syntax:

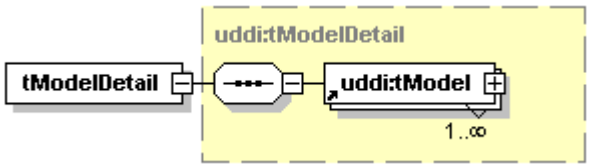


5.1.18.2 Arguments:

- ***authInfo***: This optional argument is an element that contains an authentication token. Registries that wish to restrict who can perform an inquiry in them typically require authInfo for this call.
- ***tModelKey***: One or more uddi_key values that represent UDDI assigned tModelKey values of specific instances of known tModel data.

5.1.18.3 Returns:

This API call returns a tModelDetail on successful match of the specified tModelKey values.



Attributes

Name	Use
truncated	optional

If multiple tModelKey values were passed, the results are returned in the same order as the keys passed.

If a large number of keys are passed, a node MAY truncate the result set. When this occurs, the response contains the attribute "truncated" with the value of "true".

5.1.18.4 Caveats:

If any error occurs in processing this API call, a dispositionReport structure is returned to the caller in a SOAP Fault. In addition to the errors common to all APIs, the following error information is relevant here:

- **E_invalidKeyPassed**: signifies that one of the uddi_key values passed did not match with any known tModelKey values. No partial results are returned – if any tModelKey values passed are not valid, this error is returned. The error text clearly calls out the offending key.

5.2 Publication API Set

The API calls in this section are used to publish and update information contained in a UDDI registry. According to the policy of the UDDI registry, a publisher selects a UDDI node where it will publish the information.

API calls in this section MUST all be implemented as synchronous and "atomic" from the point of view of the caller. That is, each call MUST either succeed completely or fail completely. Partial results MUST NOT be returned.

5.2.1 Publishing entities with node assigned keys

When a publisher does not provide keys for new entities, the UDDI node will assign keys in accordance with registry policy. Node-assigned keys MUST use keys that conform to the grammar in Section 4.4 *About uddiKeys*.

5.2.2 Publishing entities with publisher-assigned keys

The registry keying policy MAY allow an entity's key to be proposed by the publisher. If the publisher does not propose a key for an entity, the registry MUST assign one.

Since entity keys MUST be unique in a registry without regard to the type of entity and since registries MUST define to impose policies concerning which publishers may publish which keys, publisher-assigned keys are subject to rules that UDDI registries enforce. Behavior that ensures uniqueness across entity types (businessEntity, businessService, bindingTemplate, tModel and subscription) is REQUIRED for all registries. In this section we discuss the behavior of registries that use the recommended "uddi:" key structure. This behavior provides uniqueness and promotes interoperability among registries, while allowing various registry-specific policies to be built. Practical guidance for the use of this facility may be found in Section 9.4.2 *General Keying Policy* and Section 9.4.3 *Policy Abstractions for the UDDI keying scheme*.

5.2.2.1 Key generator keys and their partitions

To ensure that publisher-generated keys do not conflict with one another, registries assign the authority to generate keys to publishers in the following manner:

1. The conceptual space of uddiKeys is divided into non-overlapping, hierarchically arranged partitions, each of which can be associated with a publisher.
2. Only the publisher associated with a particular partition is given the authority to assign keys within the partition.
3. The publisher with authority for a given partition may designate any publisher it chooses for any partition directly below the partition it manages, provided it has not already designated a publisher to that partition.
4. The publisher with authority for a partition may transfer its authority to another publisher.
5. Initially, the registry itself has authority for the root partition of the hierarchy.

The specific mechanisms that enforce these rules are explained below.

Each node of a registry is a generator of keys. This is required to enable the node to generate keys not provided by publishers. In addition, the policies of a registry MAY allow individual publishers to obtain the authority to be generators of keys for specific partitions within the space of uddiKeys. Publishers obtain this authority by owning a particular tModel called a key

generator tModel. The key generator tModel contains a key generator key, and it specifies the partition for which the publisher may assign keys.

The subset of derivedKeys called *key generator keys* consists of all the keys of the form:

keyGeneratorKey = uddiKey ":keygenerator"

As described in Section 4.4.1, a derivedKey is one that is formed from another key by appending a non-empty, colon-prefixed string to another uddiKey. A derivedKey is said to be "based on" this uddiKey. With this in mind, the complete partition of a given keyGeneratorKey is the set of keys consisting of:

1. The set of derivedKeys based on the same uddiKey that the keyGeneratorKey is based upon.
2. The set of keyGeneratorKeys based on a key that is in the partition.
3. The domainKey, if the keyGeneratorKey is based upon that domainKey.

Note that the partition's keyGeneratorKey itself is exluded from the partition.

A rootKeyGeneratorKey is a keyGeneratorKey that is not based on a derivedKey. That is:

rootKeyGeneratorKey = (uuidKey / domainKey) ":keygenerator"

5.2.2.1.1 Examples

Based on the rules above, it is possible to construct the keyGeneratorKey for any key by manipulating the string representation of the key. To illustrate, suppose the key is x, then the following pseudo-code will determine the keyGeneratoryKey:

```
If x is a keyGeneratorKey, and y is that key minus the ":keygenerator" suffix,
then if y is a domainKey
   then x is a top-level keyGenerator, and has no keyGeneratorKey (1.a)
   else y is a derivedKey, based on z,
        and x's keyGeneratorKey is z:keyGenerator (1.b)
else
    If x is a domainKey
    then x's keyGeneratorkey is x:keyGenerator (2)
    else x is based on a key y, and x's keyGenerator is y:keyGenerator (3)
```

Using this pseudo-code illustration, the following table provides examples of legal URI's and their associated key generators for each of the four cases noted:

Key	keyGeneratorKey	Case in pseudo-code
uddi:tempuri.com	uddi:tempuri.com:keygenerator	2
uddi:tempuri.com:keygenerator	<none>	1.a
uddi:tempuri.com:xxx:keygenerator	uddi:tempuri.com:keygenerator	1.b
uddi:tempuri.com:xxx	uddi:tempuri.com:keygenerator	3
uddi:tempuri.com:xxx:yyy	uddi:tempuri.com:xxx:keygenerator	3

The following keys do NOT belong to the partition of the key generator key "uddi:tempuri.com:keygenerator".

"uddi:tempuri.com:keygenerator"	The keyGeneratorKey does not belong to the partition it designates.
"uddi:tempuri.com:xxx:yyy"	This key belongs to the partition of the keyGeneratorKey "uddi:tempuri.com:xxx:keygenerator", not this one.
"uddi:tempuri.com:keygenerator:zzz"	This key does not belong in any partition – it is an invalid key.

5.2.2.2 Behavior of publishers

To successfully publish a new entity with a proposed key, the publisher needs to own the key generator tModel for the partition in which the key lies. Typically, a publisher gets ownership by publishing the tModel in question, but publishers can also get ownership in other ways, for example by having another publisher transfer ownership.

Once a publisher owns a key generator tModel that publisher MAY publish new entities[20] and assign them keys within the key generator tModel's partition. New keys can only be generated from keyGenerator tModels that are not hidden. Publishers are responsible for managing the uniqueness of the keys in the partition they own. If a publisher fails to do so, and generates an already used key, a publish operation could inadvertently replace an entity previously published by that publisher.

If a publisher owns key generator tModels with the same key in multiple registries – for example one in the publisher's private test registry and one in the UDDI Business Registry – that publisher MAY publish the entities with identical keys in those registries. This enables many interesting capabilities. For example, publishers may choose to develop their UDDI entities by publishing them into test registries and then, at appropriate times, "promote" them to the UDDI Business Registry.

5.2.2.3 Behavior of UDDI nodes

To ensure that publisher-assigned keys work correctly all UDDI implementations behave as follows.

5.2.2.3.1 "New" and "existing" entities defined

During a publish operation, the entity or entities being published are either "new" or "existing". An existing entity is one that has a key that matches the key of an entity already in the registry. A new entity is one that does not. If a new entity has a key, this key is the key proposed for that entity by its publisher.

5.2.2.3.2 Behavior with respect to entities for which no key is proposed.

A UDDI node MUST generate and assign a key to each entity for which the publisher proposes no key. It may generate uuidKeys for use as the keys of new entities for which no key is proposed or it may generate keys in the partition of a key generator tModel it owns.

A registry whose nodes assign uddiKeys to new entities is called a root registry. The UDDI Business Registry is a root registry. A registry whose nodes gain ownership of their key

[20] Assuming, of course, the publisher is otherwise authorized to publish the entities in question. Some registries, for example, impose limits on the numbers of entities a publisher may publish.

generator tModels by publishing them in the UDDI Business Registry are affiliates of the UDDI Business Registry. See Section 1.5.5 *Affiliations of Registries*.

5.2.2.3.3 Behavior with respect to uuidKeys

A UDDI node SHOULD accept a uuidKey as the key for a new entity during a publish operation if the publisher is a trusted publisher of such keys, according to the policies of the registry. UDDI nodes MUST NOT allow other publishers to generate uuidKeys.

5.2.2.3.4 Behavior with respect to key generator keys

A UDDI node MUST NOT publish any non-tModel entity whose proposed key is a key generator key. A tModel whose proposed key is a key generator key MUST include a category bag with a keyed reference with the tModelKey set to "uddi:uddi.org:categorization:types" and the keyValue set to "keyGenerator".

5.2.2.3.5 Behavior with respect to root key generator keys

During a publish operation a UDDI node SHOULD accept a root key generator key as the key for a new tModel if it is proposed by a publisher authorized to publish the key, according to the policies of the registry. The policy MUST prevent more than one publisher from publishing tModels with the same root key generator key.

An appropriate policy for root and for affiliated registries is given in Chapter 9 *Policy*.

5.2.2.3.6 Behavior with respect to other proposed keys

A UDDI node SHOULD accept keys proposed for new entities during publishing operations if they meet both of the following criteria.

- The proposed key lies in the partition of the key of an existing key generator tModel and the key generator tModel is not hidden.
- The same publisher who is proposing the new key owns the key generator tModel referred to in the previous bullet. [21]

---

[21] The rules given in section 5.2.2.3 are safe for the following reasons. From a technical point of view the base requirement is that the keys for each entity be unique within the registry in which it is published. Because UDDI can be a replicated data store, the uniqueness requirement means that each trusted source of keys must be able to assert that the keys it publishes will not conflict with the keys published by any other trusted source. The sources in a given registry must trust one another not to generate duplicate keys, just as they do in V1 and V2. In V1 and V2 the only sources of keys are the nodes themselves.

The rootKeyGeneratorKeys come from two sources, those based on uuidKeys and those based on domainKeys. Since no domain is also a UUID, the two sources never overlap and can be considered separately.

Those rootKeyGenerators based on uuidKey>s are safe because they are generated by the nodes in the same way that <uuidKey>s are, except that the nodes append a ":keygenerator" onto the end. The <rootKeyGeneratorKey>s based on <domainKey>s are safe because they are checked for uniqueness before publishing using the UDDI registry's policy, which is required to check for uniqueness.

All the other keys are generated based on a key generator tModel. At any given time each key generator tModel is owned by a publisher and only this publisher is allowed to generate keys based on it. Further, publication may take place only on the node that has custody of the key generator tModel in question. Taken together, this means that each node can ensure for the registry as a whole that only the owner uses a given tModel to generate new keys. These features allow the nodes to apply whatever political/legal policies they wish to apply to the generation of such keys.

As for registry-wide uniqueness, any node at which a publish operation takes place can determine whether a key based on a key generator tModel is new or already existing by looking only at local information. This is obviously true for keys based on tModels that have been in the custody of the same node since their creation since all previous key creations based on the tModel in question must have been done at the custodial node. But it is also true for key generator tModels whose custody has been transferred from one node to another. This is so because any key created using the tModel must have been created prior to the custody transfer. This means that all entities with keys based on a transferred key generator tModel will precede the transfer of custody in the replication stream and thus be available at the local node.

5.2.2.4 Affiliations of registries

A set of registries may cooperate in managing a single multi-registry key space by designating one of the registries in the group to be the "root registry" and assigning it to be the authority for the root partition. Other registries in the set are said to be affiliate registries. See Section 1.5.5 *Affiliations of Registries* for more information

The UDDI Business Registry is a root registry. Its policies and procedures are designed to make it simple for any UDDI registry to be affiliated with it.

Designating new authorities is done by publishing key generator tModels in the root registry, in one or more of the registries affiliated with the root registry or both. The owner of a key generator tModel is the naming authority for the partition the tModel represents.

5.2.3 Special considerations for validated value sets

Several of the APIs defined in this section allow publishers to save category and identifier information in support of searches that use category and identifier system references. The save_business, save_service, save_binding and save_tModel APIs allow designation of these value set references. Categorization is specified using the element named categoryBag, which contains namespace-qualified references to categories and descriptions. categoryBags can also contain groups of these references to categories and descriptions. Identifiers are specified using the identifierBag element, which contains namespace-qualified references to identifiers and descriptions.

Similarly, the add_publisherAssertions and set_publisherAssertions APIs allow publisherAssertion elements to be saved. These publisherAssertion elements contain a characterization of the relationship being established using a keyedReference element that refers to a relationship type system.

Identifier, category and relationship type systems taken together are referred to as "value sets." UDDI allows value sets to be checked or unchecked. References to checked value sets that are registered in UDDI can be checked internally by the UDDI nodes where publishing takes place, or externally by a provider of a validation Web service. The UDDI node can also choose to not support some or all checked value sets.

When a UDDI node encounters a reference to a checked value set in a keyedReference it will either ensure the reference is validated or fail the save. Such references to supported checked value sets are verified for validity according to the validation algorithm defined for the value set and described by its tModel. When all checks succeed, the save is permitted. An E_unvalidatable error indicates the checked value set is supported but its validation algorithm is not available. An E_unsupported indicates the checked value set is not supported by the node. E_invalidValue or E_valueNotAllowed indicate one or more references failed validation. When the checked value set is not supported, the value set's validation algorithm is unavailable, or any of the references fail validation, the save operation MUST fail.

When the UDDI node supports a checked value set it may check the references itself, or consult a validation Web service. For cached checked value sets, the UDDI node verifies that referenced keyValues are in the set of valid values for the value set. The selection of an algorithm for verifying a checked value set is a matter of registry policy as detailed in Chapter 9 *Policy*.

A category group system is portrayed by a keyedRefererenceGroup element. Each keyedReferenceGroup has a tModelKey that references the category group system, and a set of contained keyedReference elements that make up the actual group of categories. Similar to references to checked value sets, validation is carried out for a keyedReferenceGroup if the referenced category group system is checked. Such validation entails verification that the keyedReferenceGroup is valid according to the validation algorithm described by the tModel for the category group system. Validation for a keyedReferenceGroup that references a cached checked category group system involves verification that the tModels referenced by

the contained keyedReference elements are valid for the category group system. The set of valid values for such a cacheable checked category group system is defined by the tModelKeys for the set of tModels that can participate in the group.

No validation is performed on references to unchecked value sets

5.2.4 Special considerations for the xml:lang attribute

During save_xx API calls, the name, description, address, and personName UDDI elements MAY be adorned with the xml:lang attribute to indicate the language in which their content is expressed. (See Chapter 3 *UDDI Registry Data Structures*.) When an optional xml:lang attribute is omitted from an element, no xml:lang attribute will be saved for that element.

Name elements in UDDI core data structures are frequently the main targets for sorts during UDDI inquiries. When a UDDI data structure has multiple names, sorting occurs on the first name. Care should be taken to list the primary name first when the entity is saved to ensure the proper placement of the entity in a sorted result set.

Values which can be passed in the language supplied in a save_xx API call MUST obey the recommended rules and syntax governing the xml:lang data type as defined in Section 3.3.2.3 *name*.

5.2.5 Publisher API summary

The publishing API calls are:

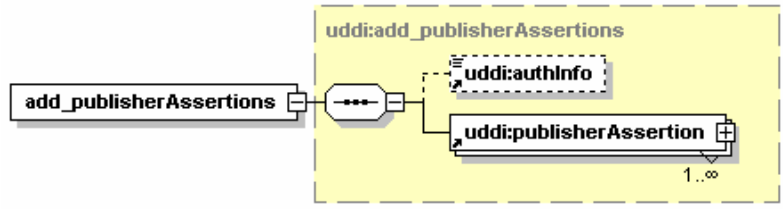
- **add_publisherAssertions**: Used to add relationship assertions to the existing set of assertions. See Appendix A *Relationships and Publisher Assertions*.
- **delete_binding**: Used to remove an existing bindingTemplate from the registry.
- **delete_business**: Used to delete existing businessEntity information from the registry.
- **delete_publisherAssertions**: Used to delete specific publisher assertions from the assertion collection controlled by a particular publisher. Deleting assertions from the assertion collection affects the visibility of business relationships. Deleting an assertion causes any relationships based on that assertion to become incomplete.
- **delete_service**: Used to delete an existing businessService from the registry.
- **delete_tModel**: Used to hide existing information about a tModel. Any tModel hidden in this way is still usable for reference purposes and accessible via the get_tModelDetail API, but is hidden from find_tModel result sets. There is no specified way to delete a tModel.
- **get_assertionStatusReport**: Used to get a status report containing publisher assertions and status information. This report is useful to help an administrator manage publisher assertions. Returns an assertionStatusReport that includes the status of all assertions made involving any businessEntity controlled by the requesting publisher.
- **get_publisherAssertions**: Used to get a list of publisher assertions that are controlled by an individual publisher. Returns a publisherAssertions structure containing all publisher assertions associated with a specific publisher.
- **get_registeredInfo**: Used to request an abbreviated list of businesses and tModels currently managed by a given publisher.
- **save_binding**: Used to register new bindingTemplate information or to update existing bindingTemplate information. Use this to control information about technical capabilities exposed by a registered business.

- **save_business**: Used to register new businessEntity information or update existing businessEntity information. Use this to control the full set of information about the entire business, including its businessService and bindingTemplate structures. This API has the broadest effect of all of the save_xx APIs.
- **save_service**: Used to register or update complete information about a businessService.
- **save_tModel**: Used to register or update information about a tModel.
- **set_publisherAssertions**: Used to save the complete set of publisher assertions for an individual publisher. Replaces any existing assertions, and causes any old assertions that are not reasserted to be removed from the registry.

5.2.6 add_publisherAssertions

The add_publisherAssertions API call causes one or more publisherAssertions to be added to an individual publisher's assertion collection. See Appendix A *Relationships and Publisher Assertions* describing relationships and the API get_publisherAssertions for more information on this collection.

5.2.6.1 Syntax:



5.2.6.2 Arguments:

- ***authInfo***: This optional argument is an element that contains an authentication token. Authentication tokens are obtained using the get_authToken API call or through some other method external to this specification. Registries that serve multiple publishers and registries that restrict who can publish in them typically require authInfo for this call.
- ***publisherAssertion***: This required repeating element holds the relationship assertions that are being added. Relationship assertions consist of a reference to two businessEntity key values as designated by the fromKey and toKey elements, as well as a REQUIRED expression of directional relationship within the contained keyedReference element. See Appendix A *Relationships and PublisherAssertions* on managing relationships. The fromKey, the toKey, and all three parts of the keyedReference – the tModelKey, the keyName, and the keyValue MUST be specified or the call will fail with the error E_fatalError. Empty (zero length) keyNames and keyValues are permitted.

5.2.6.3 Behavior:

The publisher must own the businessEntity referenced in the fromKey, the toKey, or both. If both of the businessKey values passed within an assertion are owned by the publisher, then the assertion is automatically complete and the relationship described in the assertion is visible via the find_relatedBusinesses API. To form a relationship when the publisher only owns one

of the two keys passed, the assertion MUST be matched exactly by an assertion made by the publisher who owns the other business referenced. Assertions exactly match if and only if they:

1. refer to the same businessEntity in their fromKeys;
2. refer to the same businessEntity in their toKeys;
3. refer to the same tModel in their tModelKeys;
4. have identical keyNames; and
5. have identical keyValues.

When a publisherAssertion being added references a checked relationship system using the tModelKey in the contained keyedReference, the reference MUST be checked for validity prior to completion of the add, or the node must return E_unsupported, indicating it does not support the referenced checked relationship system. Validation of a relationship system reference entails verification that the reference is valid according to the validation algorithm defined for the relationship system and described by its tModel. For cached checked relationship systems, the validation algorithm verifies that referenced keyValues are in the set of valid values for the relationship system.

For registries supporting the subscription APIs at any node, it is necessary to track a modified date for publisherAssertion elements so that nodes have the necessary information for responding to subscription requests involving find_relatedBusinesses and get_assertionStatusReport filters.

5.2.6.4 Returns:

Upon successful completion, an empty message is returned. See section 4.8 *Success and Error Reporting.*

5.2.6.5 Caveats:

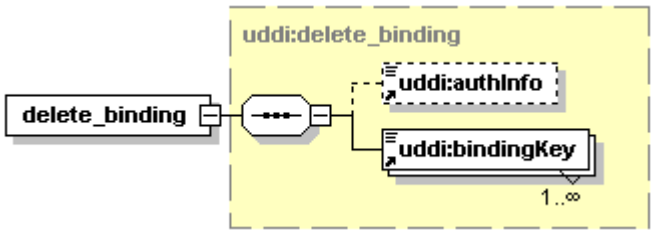
If an error occurs in processing this API call, a dispositionReport structure MUST be returned to the caller in a SOAP Fault. See Section 4.8 *Success and Error Reporting.* In addition to the errors common to all APIs, the following error information is relevant here:

- **E_invalidKeyPassed**: signifies that one of the *uddiKey* values passed did not match with any known businessKey or tModelKey values. The key and element or attribute that caused the problem SHOULD be clearly indicated in the error text.
- **E_userMismatch**: signifies that neither of the businessKey values passed in the embedded fromKey and toKey elements is owned by the publisher associated with the authentication token. The error text SHOULD clearly indicate which assertion caused the error.

5.2.7 delete_binding

The delete_binding API call causes one or more instances of bindingTemplate data to be deleted from the UDDI registry.

5.2.7.1 Syntax:



5.2.7.2 Arguments:

- ***authInfo***: This optional argument is an element that contains an authentication token. Authentication tokens are obtained using the get_authToken API call or through some other method external to this specification. Registries that serve multiple publishers and registries that restrict who can publish in them typically require authInfo for this call.
- ***bindingKey***: One or more required *uddiKey* values that represent specific instances of known bindingTemplate data.

5.2.7.3 Returns:

Upon successful completion, an empty message is returned. See section 4.8 *Success and Error Reporting.*

5.2.7.4 Caveats:

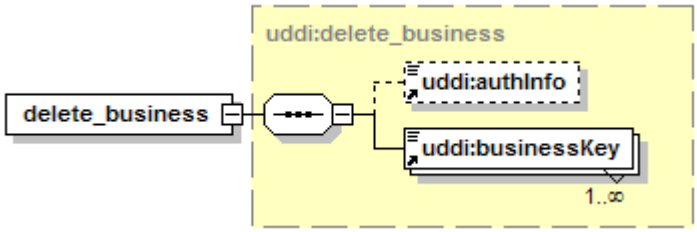
If an error occurs in processing this API call, a dispositionReport structure MUST be returned to the caller in a SOAP Fault. See Section 4.8 *Success and Error Reporting.* In addition to the errors common to all APIs, the following error information is relevant here:

- **E_invalidKeyPassed**: Signifies that one of the *uddiKey* values passed did not match with any known bindingKey values or multiple instances of the same bindingKey values were passed. No partial results are returned – if any bindingKey values passed are not valid, this error is returned. The key that caused the problem SHOULD clearly be indicated in the error text.
- **E_userMismatch**: Signifies that one or more of the bindingKey values passed refers to a bindingTemplate that is not owned by the individual publisher associated with the authentication token.

5.2.8 delete_business

The delete_business API call is used to remove one or more business registrations and all elements that correspond to the natural content of the corresponding businessEntity elements from a UDDI registry.

5.2.8.1 Syntax:



5.2.8.2 Arguments:

- ***authInfo***: This optional argument is an element that contains an authentication token. Authentication tokens are obtained using the get_authToken API call or through some other means external to this specification. Registries that serve multiple publishers and registries that restrict who can publish in them typically require authInfo for this call.
- ***businessKey***: One or more required *uddiKey* values that represent specific instances of known businessEntity data.

5.2.8.3 Behavior:

The UDDI registry MUST permanently remove all of the *natural contents*[22] of the passed businessEntity elements, including any currently nested businessService and bindingTemplate data, from the UDDI registry.

If there are service projections[23] that reference businessService elements deleted in this way, they are left untouched. Such "broken" service projections appear in their businessEntity as businessService elements containing the businessKey and serviceKey attributes as their only content. For this reason, it is a best practice to coordinate references to businessService data published under another businessEntity with the party who manages that data.

All publisher assertions that reference the businessKey of the businessEntity being deleted in either the fromKey or toKey of the publisherAssertion MUST be automatically deleted. A deleted business MUST not be returned in the find_relatedBusinesses API.

Any transferToken referring to the business entity being deleted becomes invalid and can no longer be used to transfer any entities.

---

[22] When a business registration is first saved, all of the contained data found in the registered businessEntity element is referred to as the natural contents. UDDI defines several types of referencing mechanisms – and referenced elements are not considered to be natural contents of registered businessEntity data. Natural contents can be recognized by matching businessKey values between businessService and businessEntity data.

[23] UDDI allows save_business API calls to be processed that contain businessService data references that are the natural children of a different business registration. Doing this creates a businessService projection that results in the data associated with the referenced businessService to be projected as though it were contained in the referencing businessEntity. These are called service projections.

5.2.8.4 Returns:

Upon successful completion, an empty message is returned. See section 4.8 *Success and Error Reporting.*

5.2.8.5 Caveats:

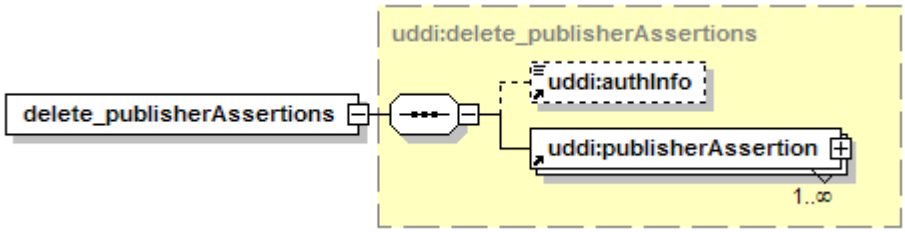
If an error occurs in processing this API call, a dispositionReport element MUST be returned to the caller within a SOAP Fault. See Section 4.8 *Success and Error Reporting.* In addition to the errors common to all APIs, the following error information is relevant here:

- **E_invalidKeyPassed**: Signifies that one of the *uddiKey* values passed did not match with any known businessKey values or multiple instances of the same businessKey values were passed. The key that caused the error SHOULD be clearly indicated in the error text.
- **E_userMismatch**: Signifies that one or more of the businessKey values passed refers to data that is not owned by the individual publisher who is represented by the authentication token.

5.2.9 delete_publisherAssertions

The delete_publisherAssertions API call causes one or more publisherAssertion elements to be removed from a publisher's *assertion collection*. See Appendix A *Relationships and Publisher Assertions* and the API get_publisherAssertions for more information on this collection.

5.2.9.1 Syntax:



5.2.9.2 Arguments:

- ***authInfo***: This optional argument is an element that contains an authentication token. Authentication tokens are obtained using the get_authToken API call or through some other means external to this specification. Registries that serve multiple publishers and registries that restrict who can publish in them typically require authInfo for this call.
- ***publisherAssertion***: One or more required publisher assertion structures exactly matching an existing assertion in the publisher's assertion collection.

5.2.9.3 Behavior:

The UDDI registry scans the assertion collection associated with the publisher, and removes any assertions that exactly match all parts of each publisherAssertion passed. Any assertions described that cannot be located result in an error. The removal of assertions in this API causes the corresponding relationships to no longer be visible via the find_relatedBusinesses API.

For registries supporting the subscription APIs at any node, it is necessary to track a modified date for publisherAssertion elements so that nodes have the necessary information for responding to subscription requests involving find_relatedBusinesses and get_assertionStatusReport filters.

5.2.9.4 Returns:

Upon successful completion, an empty message is returned. See section 4.8 *Success and Error Reporting.*

5.2.9.5 Caveats:

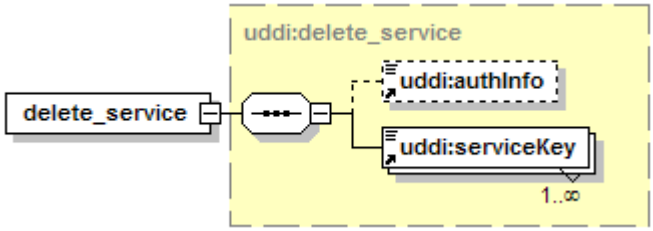
If an error occurs in processing this API call, a dispositionReport structure MUST be returned to the caller in a SOAP Fault. See Section 4.8 *Success and Error Reporting.* In addition to the errors common to all APIs, the following error information is relevant here:

- **E_assertionNotFound**: Signifies that one of the assertion structures passed does not have any corresponding match in the publisher's assertion collection or multiple instances of the same publisherAssertions elements were passed. The assertion that caused the problem SHOULD be clearly indicated in the error text.

5.2.10 delete_service

The delete_service API call is used to remove one or more businessService elements from the UDDI registry and from its containing businessEntity parent.

5.2.10.1 Syntax:



5.2.10.2 Arguments:

- ***authInfo***: This optional argument is an element that contains an authentication token. Authentication tokens are obtained using the get_authToken API call or through some other means external to this specification. Registries that serve multiple publishers and registries that restrict who can publish in them typically require authInfo for this call.
- ***serviceKey***: One or more required *uddiKey* values that represent specific instances of known businessService data.

5.2.10.3 Behavior:

All contained bindingTemplate data MUST also be removed from the registry as a result of this call.

If a business service being deleted is the target of a business service projection associated with another businessEntity, the referencing businessService elements are left untouched. Such "broken" service projections appear in their businessEntity and businessService elements containing the businessKey and serviceKey attributes as their only content. For this reason, it is recommended that references to businessService data published under another businessEntity be coordinated with the party that manages that data.

5.2.10.4 Returns:

Upon successful completion, an empty message is returned. See section 4.8 *Success and Error Reporting.*

5.2.10.5 Caveats:

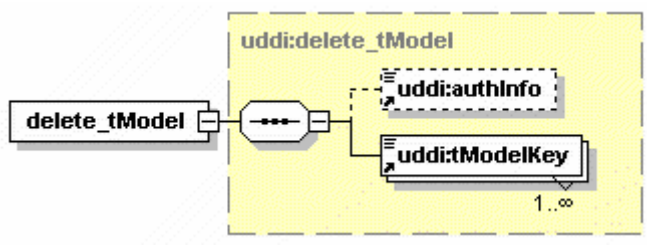
If an error occurs in processing this API call, a dispositionReport structure MUST be returned to the caller in a SOAP Fault. See Section 4.8 *Success and Error Reporting.* In addition to the errors common to all APIs, the following error information is relevant here:

- **E_invalidKeyPassed**: Signifies that one of the *uddiKey* values passed did not match with any known serviceKey values or multiple instances of the same serviceKey values were passed. The key causing the error SHOULD be clearly indicated in the error text.
- **E_userMismatch**: Signifies that one or more of the serviceKey values passed refers to data that is not owned by the individual publisher who is represented by the authentication token.

5.2.11 delete_tModel

The delete_tModel API call is used to logically delete one or more tModel structures. Logical deletion hides the deleted tModels from find_tModel result sets but does not physically delete them. New references to deleted (hidden) tModels can be established by publishers that know their keys. Deleting an already deleted tModel has no effect.

5.2.11.1 Syntax:



5.2.11.2 Arguments:

- ***authInfo***: This optional argument is an element that contains an authentication token. Authentication tokens are obtained using the get_authToken API call or through some other means external to this specification. Registries that serve multiple publishers and registries that restrict who can publish in them typically require authInfo for this call.
- ***tModelKey***: One or more required *uddiKey* values that represent specific instances of known tModel data.

5.2.11.3 Behavior:

If a tModel is hidden in this way it MUST not be physically deleted as a result of this call. Any tModels hidden in this way are still accessible, via the get_registeredInfo and get_tModelDetail APIs, but are omitted from any results returned by calls to find_tModel. All other inquiry APIs may include references to tModelKeys of deleted tModelKeys, and UDDI data structures that reference these tModels are found and retrieved.

The purpose of the delete_tModel behavior is to ensure that the details associated with a hidden tModel are still available to anyone currently using the tModel. A hidden tModel can be restored and made visible to search results by invoking the save_tModel API at a later time, passing the original data and the tModelKey value of the hidden tModel.

It is not an error to transfer a hidden tModel (i.e. deleted attribute set to TRUE).

5.2.11.4 Returns:

Upon successful completion, an empty message is returned. See section 4.8 *Success and Error Reporting.*

5.2.11.5 Caveats:

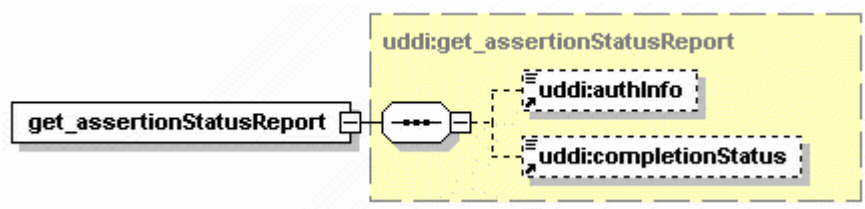
If an error occurs in processing this API call, a dispositionReport element MUST be returned to the caller within a SOAP Fault. See Section 4.8 *Success and Error Reporting.* In addition to the errors common to all APIs, the following error information is relevant here:

- **E_invalidKeyPassed**: Signifies that one of the *uddiKey* values passed did not match with any known tModelKey values or multiple instances of the same tModelKey values were passed. The invalid key references SHOULD be clearly indicated in the error text.
- **E_userMismatch**: Signifies that one or more of the tModelKey values passed refers to data that is not owned by the individual publisher who is represented by the authentication token.

5.2.12 get_assertionStatusReport

The get_assertionStatusReport API call provides administrative support for determining the status of current and outstanding publisher assertions that involve any of the business registrations managed by the individual publisher. Using this API, a publisher can see the status of assertions that they have made, as well as see assertions that others have made that involve businessEntity structures controlled by the requesting publisher. See Appendix A *Relationships and Publisher Assertions* for more information.

5.2.12.1 Syntax:



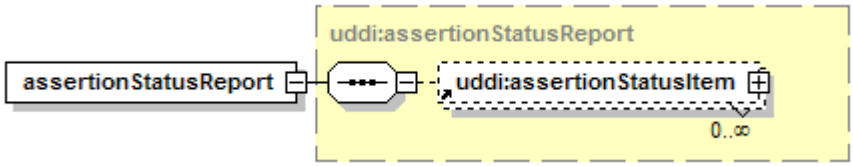
5.2.12.2 Arguments:

- ***authInfo***: This optional argument is an element that contains an authentication token. Authentication tokens are obtained using the get_authToken API call or through some other means external to this specification. Registries that serve multiple publishers and registries that restrict who can publish in them typically require authInfo for this call.
- ***completionStatus***: This optional argument lets the publisher restrict the result set to only those relationships that have the specified status value. Assertion status is a calculated result based on the sum total of assertions made by the individuals that control specific business registrations. When no completionStatus element is provided, all assertions involving the businesses that the publisher owns are retrieved, without regard to the completeness of the relationship. completionStatus MUST contain one of the following values
    - **status:complete**: Passing this value causes only the publisher assertions that are complete to be returned. Each businessEntity listed in assertions that are complete has a visible relationship that directly reflects the data in a complete assertion (as described in the find_relatedBusinesses API).
    - **status:toKey_incomplete**: Passing this value causes only those publisher assertions where the party who controls the businessEntity referenced by the toKey value in an assertion, has not made a matching assertion, to be listed.
    - **status:fromKey_incomplete**: Passing this value causes only those publisher assertions where the party who controls the businessEntity referenced by the fromKey value in an assertion, has not made a matching assertion, to be listed.
    - **status:both_incomplete**. This status value, however, is only applicable to the context of UDDI subscription and SHOULD not be present as part of a response to a get_assertionStatusReport request.

5.2.12.3 Returns:

Upon successful completion, an assertionStatusReport structure is returned containing zero or more assertionStatusItem structures. Elements will be sorted by last date change in ascending order.

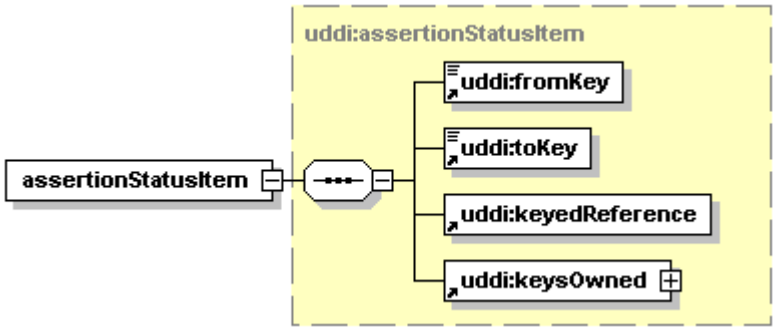
The assertionStatusReport has the form:



The assertionStatusReport reports all complete and incomplete assertions and serves an administrative use for determining if there are any outstanding, incomplete assertions pertaining to relationships involving businesses with which the publisher is associated.

Since the publisher who was authenticated by the get_assertionStatusReport API may own several businesses, the assertionStatusReport structure shows the assertions made for all businesses owned by the publisher.

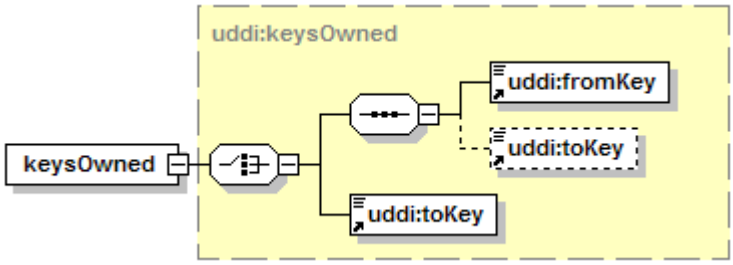
The assertion status report is composed of a set of assertionStatusItem elements that describe the assertions in which the publisher's businesses participate. The assertionStatusItem element has the form:



The assertionStatusItem structure has the following attribute:

Name	Use
completionStatus	required

While the elements fromKey, toKey and keyedReference together identify the assertion on whose status a report is being provided, the keysOwned element designates those businessKeys the publisher manages. The keysOwned element has the form:



An assertion is part of a reciprocal relationship only if the completionStatus attribute has a value "status:complete". If completionStatus has a value "status:toKey_incomplete" or "status:fromKey_incomplete", the party who controls the businessEntity referenced by the toKey or the fromKey has not yet made a matching assertion.

5.2.12.4 Caveats:

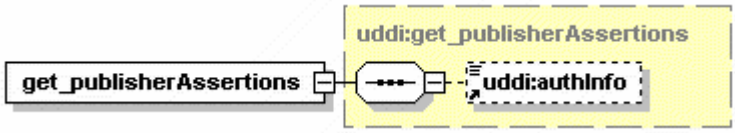
If an error occurs in processing this API call, a dispositionReport element MUST be returned to the caller within a SOAP Fault. See Section 4.8 *Success and Error Reporting.* In addition to the errors common to all APIs, the following error information is relevant here:

- **E_invalidCompletionStatus**: Signifies that the completionStatus value passed is unrecognized. The completion status that caused the problem SHOULD be clearly indicated in the error text.

5.2.13 get_publisherAssertions

The get_publisherAssertions API call is used to obtain the full set of publisher assertions that is associated with an individual publisher. It complements the get_registeredInfo API which returns information about businesses, services, bindings, and tModels managed by a publisher.

5.2.13.1 Syntax:



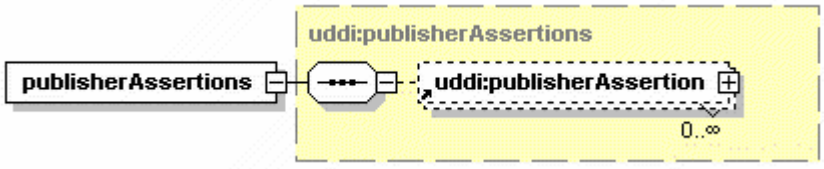
5.2.13.2 Arguments:

- ***authInfo***: This optional argument is an element that contains an authentication token. Authentication tokens are obtained using the get_authToken API call or through some other means external to this specification. Registries that serve multiple publishers and registries that restrict who can publish in them typically require authInfo for this call.

5.2.13.3 Returns:

This API call returns a publisherAssertions structure that contains a publisherAssertion element for each publisher assertion registered by the publisher. When the registry distinguishes between publishers, this information is associated with the authentication information. Only assertions made by the publisher are returned. Elements will be sorted by last date change in ascending order. See get_assertionStatusReport and Appendix A *Relationships and Publisher Assertions* for more details.

The publisherAssertions structure has the form:



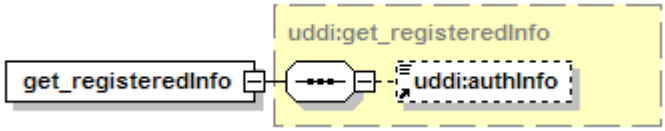
5.2.13.4 Caveats:

None, other than those common to all UDDI APIs. See Section 12.1 *Common Error Codes.*

5.2.14 get_registeredInfo

The get_registeredInfo API call is used to get an abbreviated list of all businessEntity and tModel data that are controlled by a publisher. When the registry distinguishes between publishers, this is the individual associated with the credentials passed in the authInfo element. This returned information is intended, for example, for driving tools that display lists of registered information and then provide drill-down features. This is the recommended API to use after a network problem results in an unknown status of saved information.

5.2.14.1 Syntax:



Attributes

Name	Use
infoSelection	required

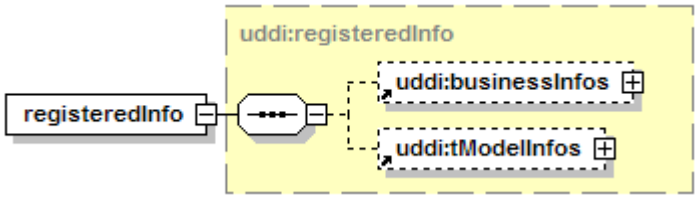
5.2.14.2 Arguments:

- ***authInfo***: This optional argument is an element that contains an authentication token. Authentication tokens are obtained using the get_authToken API call or through some other means external to this specification. Registries that serve multiple publishers and registries that restrict who can publish in them typically require authInfo for this call.
- ***infoSelection***: This required argument represents an enumerated choice that determines which tModels are returned. "all" indicates all visible and hidden tModels owned by the publisher are to be returned (this is the default). "visible" indicates only visible tModels owned by the publisher are to be returned. "hidden" indicates only hidden (logically deleted) tModels owned by the publisher are to be returned.

5.2.14.3 Returns:

Upon successful completion, a registeredInfo structure MUST be returned, listing abbreviated business information in one or more businessInfo elements, and tModel information in one or more tModelInfo elements. This API is useful for determining the full extent of registered business and tModel information owned by a single publisher in a single call. This structure complements the get_publisherAssertions API call, which returns information about assertions owned by an individual publisher. businessInfos and/or tModelInfos will be sorted case-sensitively on the primary name in ascending order, using the collation sequence determined by node policy.

The registeredInfo structure has the form:



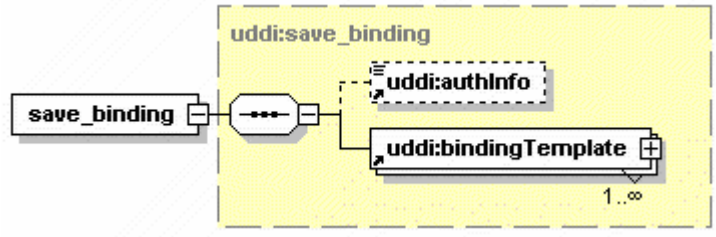
5.2.14.4 Caveats:

None, other than those common to all UDDI APIs. See Section 12.1 *Common Error Codes.*

5.2.15 save_binding

The save_binding API call is used to save or update a complete bindingTemplate element. It can be used to add or update one or more bindingTemplate elements as well as the container/contained relationship that each bindingTemplate has with one or more existing businessService elements. Each bindingTemplate MAY be signed and MAY have publisher-assigned keys.

5.2.15.1 Syntax:



5.2.15.2 Arguments:

- ***authInfo***: This optional argument is an element that contains an authentication token. Authentication tokens are obtained using the get_authToken API call or through some other means external to this specification. Registries that serve multiple publishers and registries that restrict who can publish in them typically require authInfo for this call.
- ***bindingTemplate***: Required repeating element containing one or more complete bindingTemplate structures. To save a new bindingTemplate, a bindingTemplate element is passed with either an empty bindingKey attribute value, or with a publisher-assigned bindingKey. See Section 5.2.2.2 *Behavior of Publishers*.

5.2.15.3 Behavior

Each new bindingTemplate passed MUST contain a serviceKey value that corresponds to a registered businessService controlled by the same publisher. An existing binding template MAY contain a serviceKey value that corresponds to a registered businessService controlled by the same publisher. The net effect of this call is to determine the containing parent businessService for each bindingTemplate affected by this call. If the same bindingTemplate (determined by matching bindingKey value) is listed more than once, any relationship to the containing businessService is determined by processing order, which is determined by the position of the bindingTemplate data in first to last order.

If the bindingKey within a bindingTemplate element is missing or is passed with an empty value, this is a signal that the bindingTemplate is being inserted for the first time. When this occurs, the node MUST automatically generate a new key for the bindingTemplate that is without an associated key. New bindingTemplate structures can also be added with publisher-assigned keys. See Section 5.2.2.2 *Behavior of Publishers*.

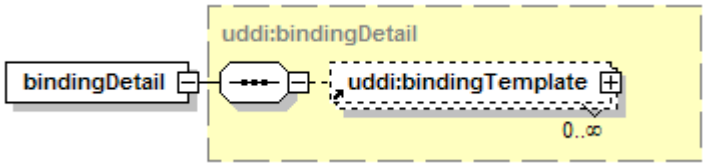
Using this API call it is possible to move an existing bindingTemplate from one businessService to another by simply specifying a different parent businessService relationship along with the complete bindingTemplate. Changing a parent relationship in this way causes two businessService structures to be affected. The net result of such a move is that the bindingTemplate still resides within one, and only one businessService based on the value of the serviceKey passed. An attempt to move a bindingTemplate in this manner by a party who is not the publisher of the businessService that is specified by the serviceKey MUST be rejected with an error E_userMismatch.

When a bindingTemplate is saved with a categoryBag content that is associated with a checked value set or category group system tModel, the references MUST be checked for validity prior to completion of the save, or the node must return E_unsupported, indicating it does not support the referenced checked value set or category group system. See Section 5.2.3 *Special considerations for validated value sets* and Appendix F *Using Categorization* for additional details.

5.2.15.4 Returns:

This API returns a bindingDetail structure containing the results of the call that reflects the newly registered information for the effected bindingTemplate elements. If more than one bindingTemplate is saved in a single save_binding call, the resulting bindingDetail MUST return results in the same order that they appeared in the save_binding call. If the same bindingTemplate (determined by matching bindingKey) is listed more than once in the save_binding call, it MAY be listed once in the result for each appearance in the save_binding call. If the same bindingTemplate appears more than once in the response, the last occurrence of the bindingTemplate in the results represents the state stored in the registry. Any bindingKeys that were assigned as a result of processing the save_binding call are included in the bindingTemplate data.

The bindingDetail structure has the form:



5.2.15.5 Caveats:

If an error occurs in processing this API call, a dispositionReport element MUST be returned to the caller in a SOAP Fault. See Section 4.8 *Success and Error Reporting.* In addition to the errors common to all APIs, the following error information is relevant here:

- **E_accountLimitExceeded**: Signifies that user account limits have been exceeded.
- **E_invalidKeyPassed**: Signifies that the request cannot be satisfied because one or more *uddiKey* values specified are not valid key values for the entities being published. tModelKey, serviceKey, or bindingKey values that either do not exist, or exist with a different entity type, or are not authorized to be proposed by the publisher are considered to be invalid values. The key causing the error SHOULD be clearly indicated in the error text. This error code will also be returned in the event that the serviceKey is not provided and the bindingKey is either absent or has a value not

registered in the registry. In this case, the error text SHOULD clearly indicate the use of an incomplete bindingTemplate.

- **E_invalidValue**: A value that was passed in a keyValue attribute did not pass validation. This applies to checked value sets that are referenced using keyedReferences. The error text SHOULD clearly indicate the key and value combination that failed validation.
- **E_keyUnavailable**: Indicates that the proposed key has already been assigned to some other publisher or is not within the partition defined by a key generator tModel that the publisher owns.
- **E_requestTimeout**: Signifies that the request could not be carried out because a needed validate_values service did not respond in a reasonable amount of time. Details identifying the failing Web service SHOULD be included in the dispositionReport element.
- **E_userMismatch**: Signifies that one or more of the *uddiKey* values passed refers to data that is not owned by the individual publisher who is represented by the authentication token.
- **E_valueNotAllowed**: Restrictions have been placed by the value set provider on the types of information that should be included at that location within a specific value set. A validate_values Web service chosen by the UDDI node has rejected this businessEntity for at least one specified keyedReference. The error text SHOULD clearly indicate the keyedReference that was not successfully validated.

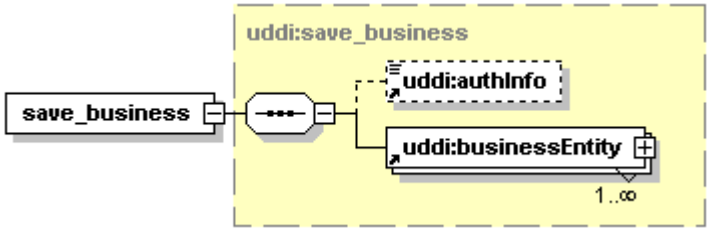
5.2.16 save_business

The save_business API call is used to save or update information about a complete businessEntity structure. This API has the broadest scope of all of the save_xx API calls, and can be used to make sweeping changes to the published information for one or more businessEntity elements controlled by an individual.

This API call can be used to establish a reference relationship to businessService structures that are managed as the contents of another businessEntity. In this way, a businessService that is a natural part of one businessEntity can appear as a *projected service* of another businessEntity. The content of a businessService projected in this way (by way of a reference established by this API) are not managed as a part of the referencing entity.

businessEntity structures MAY be signed and MAY have publisher-assigned keys.

5.2.16.1 Syntax:



5.2.16.2 Arguments:

- ***authInfo***: This optional argument is an element that contains an authentication token. Authentication tokens are obtained using the get_authToken API call or through some other means external to this specification. Registries that serve multiple publishers and registries that restrict who can publish in them typically require authInfo for this call.
- ***businessEntity***: Required repeating element containing one or more businessEntity structures. These can be obtained in advance by using the get_businessDetail API call or by any other means.

5.2.16.3 Behavior:

If any of the *uddiKey* values within a businessEntity element (e.g. any data with a key value regulated by a businessKey, serviceKey or bindingKey) is missing or is passed with an empty value, this is a signal that the data that is so keyed is being inserted for the first time.[24] When this occurs, the node MUST automatically generate a new key for the data passed that is without an associated key. New entities can also be added with publisher-assigned keys. See Section 5.2.2.2 *Behavior of Publishers*.

To make this API call perform an update to existing registered data, the keyed entities (businessEntity, businessService or bindingTemplate) MUST have *uddiKey* values that correspond to the registered data to be updated.

[24] This does not apply to structures that use keys to refer to other keyed data, such as tModelKey references within bindingTemplate or keyedReference structures, since these are references. These keys MUST contain values that refer to actual entities.

Data can be deleted with this API call when registered information is different from the new information provided. Any businessService and bindingTemplate structures found in the custodial UDDI node, but missing from the businessEntity information provided in this call, are deleted from the registry by this call.

Data contained within businessEntity structures can be rearranged with this API call. This can be done by redefining parent container relationships for other registered information. For instance, if a new businessEntity is saved with information about a businessService that is registered already as part of a different businessEntity, this results in the businessService being moved from its current container to the new businessEntity. This condition occurs when the businessKey of the businessService being saved matches the businessKey of the businessEntity being saved. An attempt to delete or move a businessService in this manner by a party who is not the publisher of the businessService MUST be rejected with an error E_userMismatch.

If the businessEntity being saved contains a businessService that has a businessKey referring to some businessEntity other than the businessEntity being saved, the UDDI registry notes a reference, called a "service projection", to the existing businessService. Subsequent calls to the get_businessDetail API, passing either the businessKey of the businessEntity that contains the referenced businessService or the businessKey of the businessEntity that contains the service projection will result in an identical businessService element being included as part of the result set.

A businessEntity must not contain a businessService and a service projection to this businessService. As a result, a businessService cannot be moved to a businessEntity that already has a service projection to that businessService. Regardless of the order of operation, a businessService and a service projection can never appear under the same businessEntity. Implementations are required to reject and return an E_fatalError during such a save_business operation.

No changes to the referenced businessService are effected by the act of establishing a service projection. Existing service projections associated with the businessEntity being saved that are not contained in the call to save_business are deleted automatically. This reference deletion does not cause any changes to the referenced businessService. If the referenced businessService is deleted by any means, all references to it associated with other businessEntity structures are left untouched. Such "broken" service projections appear in their businessEntity as businessService elements containing the businessKey and serviceKey attributes as their only content. If the businessService is moved to another business, all projections will be updated to reflect the new businessKey[25]. For this reason, it is good practice to coordinate references to businessService data published under another businessEntity with the party who manages that data

When saving a businessEntity containing a service projection, all of the content of the businessService provided in the save_business, with the exception of the serviceKey and businessKey, is ignored. The businessKey and serviceKey of the businessService being referenced are used to determine if the businessService is for a service projection or not. If the businessService identified by the serviceKey is not part of the businessEntity identified by the businessKey, the error E_invalidProjection will be returned.

When a businessEntity is saved with identifierBag or categoryBag contents that is associated with a checked value set or category group system tModel, the references MUST be checked for validity prior to completion of the save or the node must return E_unsupported, indicating it does not support the referenced checked value set or category group system. See Section

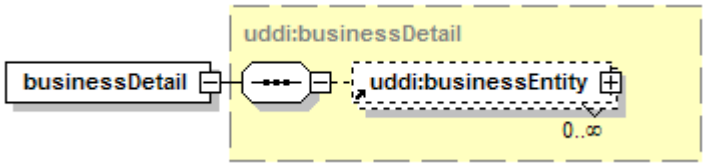
[25] This solution emphasizes that in all circumstances the real service should be used to retrieve the properties within a service projection. Based on this premise, there should be no validation or ownership verification done on the businessKey associated with a service during replication.

5.2.3 *Special considerations for validated value sets,* Appendix E *Using Identifiers* and Appendix F *Using Categorization* for additional details.

5.2.16.4 Returns:

This API returns a businessDetail structure containing the final results of the call that reflects the new registered information for the businessEntity information provided. Any businessKey, serviceKey, or bindingKey attributes that were assigned as a result of processing the save_business are included in the returned data. For businessService elements that are service projections, the response includes either the businessService elements as provided by the publisher or the full contents of the real businessService elements. These results include any businessService elements that are contained by reference. If the same entity (businessEntity, businessService, or bindingTemplate), determined by matching key, is listed more than once in the save_business call, it MAY be listed once in the result for each appearance in the call. If the same entity appears more than once in the response, the last appearance occurrence of the entity in the results represents either the final saved state stored in the registry or the last occurrence of the entity provided by the publisher within the request.

The businessDetail has the form:



5.2.16.5 Caveats

If an error occurs in processing this API call, a dispositionReport element MUST be returned to the caller in a SOAP Fault. See Section 4.8 *Success and Error Reporting.* In addition to the errors common to all APIs, the following error information is relevant here:

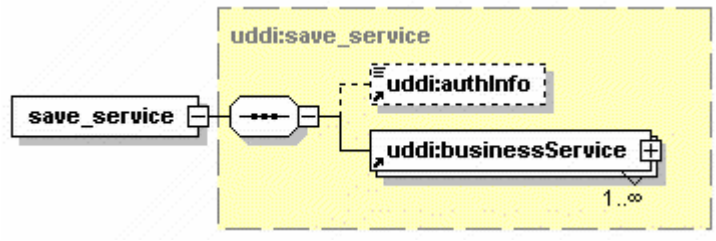
- **E_accountLimitExceeded**: Signifies that user account limits have been exceeded.
- **E_invalidKeyPassed**: Signifies that the request cannot be satisfied because one or more *uddiKey* values specified are not valid key values for the entities being published. tModelKey, businessKey, serviceKey, or bindingKey values that either do not exist, or exist with a different entity type, or are not authorized to be proposed by the publisher are considered to be invalid values. The key causing the error SHOULD be clearly indicated in the error text.
- **E_invalidProjection**: Signifies that an attempt was made to save a businessEntity containing a service projection where the businessService being projected is not a part of the businessEntity that is identified by the businessKey in the businessService. The serviceKey of at least one such businessService SHOULD be included in the dispositionReport.
- **E_userMismatch**: Signifies that one or more of the *uddiKey* values passed refers to data that is not owned by the individual publisher who is represented by the authentication token. The key causing the error SHOULD be clearly indicated in the error text.
- **E_invalidValue**: A value that was passed in a keyValue attribute did not pass validation. This applies to checked value sets that are referenced using keyedReferences. The error text SHOULD clearly indicate the key and value combination that failed validation.

- **E_keyUnavailable**: Indicates that the proposed key has already been assigned to some other publisher or is not within the partition defined by a key generator tModel that the publisher owns.
- **E_requestTimeout**: Signifies that the request could not be carried out because a needed validate_values service did not respond in a reasonable amount of time. Details identifying the failing Web service SHOULD be included in the dispositionReport element.
- **E_unsupported**: A keyedReference in a categoryBag or an identifierBag that references a checked value set cannot be validated by the UDDI node because the node does not support the referenced checked value set. The error text should clearly indicate the keyedReference that cannot be validated.
- **E_unvalidatable**: A keyedReference in a categoryBag or an identifierBag that references a checked value set cannot be validated by the UDDI node because the referenced tModel has been marked unvalidatable. The error text should clearly indicate the keyedReference that cannot be validated.
- **E_valueNotAllowed**: Restrictions have been placed by the value set provider on the types of information that should be included at that location within a specific value set. A validate_values Web service chosen by the UDDI node has rejected this businessEntity for at least one specified keyedReference. The error text SHOULD clearly indicate the keyedReference that was not successfully validated.

5.2.17 save_service

The save_service API call adds or updates one or more businessService elements. Each businessService MAY be signed and MAY have publisher-assigned keys.

5.2.17.1 Syntax:



5.2.17.2 Arguments:

- ***authInfo***: This optional argument is an element that contains an authentication token. Authentication tokens are obtained using the get_authToken API call or through some other means external to this specification. Registries that serve multiple publishers and registries that restrict who can publish in them typically require authInfo for this call.
- ***businessService***: Required repeating element containing one or more complete businessService elements. For the purpose of performing round trip updates, this data can be obtained in advance by using the get_serviceDetail API call or by any other means.

5.2.17.3 Behavior:

Each new businessService passed MUST contain a businessKey value that corresponds to a registered businessEntity controlled by the same publisher. An existing business service MAY contain a businessKey value that corresponds to a registered businessEntity controlled by the same publisher.

If any of the *uddiKey* values within a businessService element (i.e., any data with a key value regulated by a serviceKey or bindingKey) is passed with an empty value, this is a signal that the data that is so keyed is being inserted for the first time. [26] In this case, a new key value MUST be automatically generated for the data which was passed without an associated key value. New entities can also be added with publisher-assigned keys. See Section 5.2.2.2 *Behavior of Publishers*.

If the same businessService is contained in more than one businessService argument, the final relationship to the containing businessEntity is determined by processing order – which is determined by first to last order of the information passed in the request. Analogously, if the same bindingTemplate is specified in the call as being in more than one businessService, the businessService that is its container at the conclusion of the call is last one listed.

[26] This does not apply to structures that use keys to refer to other keyed data, such as tModelKey references within bindingTemplate or keyedReference structures, since these are references. These keys MUST contain values that refer to actual entities.

Using this API call it is possible to move an existing bindingTemplate element from one businessService element to another, or move an existing businessService element from one businessEntity to another by simply specifying a different parent businessEntity relationship. Changing a parent relationship in this way causes two businessEntity or two businessService structures to be changed. An attempt to move a bindingTemplate or a businessService in this manner by a party who is not the publisher of the businessService that is specified by the serviceKey or the businessEntity that is specified by the businessKey MUST be rejected with an error E_userMismatch.

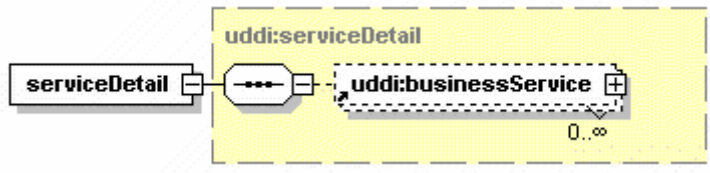
When a businessService is saved with categoryBag contents that is associated with a checked value set or category group system tModel, the references MUST be checked for validity prior to completion of the save or the node MUST return E_unsupported, indicating it does not support the referenced checked value set or category group system. See Section 5.2.3 *Special considerations for validated value sets* and Appendix F *Using Categorization* for additional details.

5.2.17.4 Returns:

This API call returns a serviceDetail containing the final results of the call that reflects the newly registered information for the affected businessService elements. In cases where multiple businessService elements are passed in the request, the result contains the final results for each businessService passed and these appear in the same order as found in the request. Any serviceKey and bindingKey values that were assigned as a result of processing the save_service API are included in the businessService data.

If the same entity (businessService, or bindingTemplate), determined by matching key, is listed more than once in the save_service API, it MAY be listed once in the result for each appearance in the save_service API. If the same entity appears more than once in the response, the last occurrence of the entity in the results represents the state stored in the registry.

The serviceDetail has the form:



5.2.17.5 Caveats:

If an error occurs in processing this API call, a dispositionReport element MUST be returned to the caller within a SOAP Fault. See Section 4.8 *Success and Error Reporting.* In addition to the errors common to all APIs, the following error information is relevant here:

- **E_accountLimitExceeded**: Signifies that user account limits have been exceeded.
- **E_invalidKeyPassed**: Signifies that the request cannot be satisfied because one or more *uddiKey* values specified are not valid key values for the entities being published. tModelKey, businessKey, serviceKey, or bindingKey values that either do not exist, or exist with a different entity type, or are not authorized to be proposed by the publisher are considered to be invalid values. The key causing the error SHOULD be clearly indicated in the error text. This error code will also be returned in the event that the businessKey is not provided and the serviceKey is either absent or has a

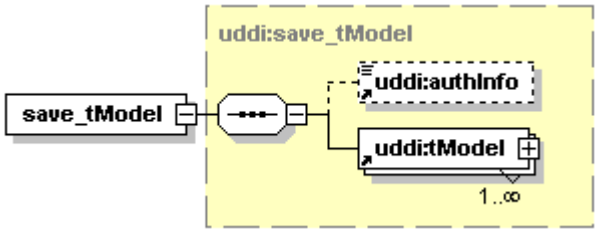
value not registered in the registry. In this case, the error text SHOULD clearly indicate the use of an incomplete businessService.

- **E_invalidValue**: A value that was passed in a keyValue attribute did not pass validation. This applies to checked value sets referenced using keyedReferences. The error text SHOULD clearly indicate the key and value combination that failed validation.
- **E_keyUnavailable**: Indicates that the proposed key has already been assigned to some other publisher or is not within the partition defined by a key generator tModel that the publisher owns.
- **E_requestTimeout**: Signifies that the request could not be carried out because a needed validate_values service did not respond in a reasonable amount of time. Details identifying the failing Web service SHOULD be included in the dispositionReport element.
- **E_userMismatch**: Signifies that one or more of the *uddiKey* values passed refers to data that is not owned by the individual publisher who is represented by the authentication token.
- **E_unsupported**: A keyedReference in a categoryBag that references a checked value set cannot be validated by the UDDI node because the node does not support the referenced checked value set. The error text SHOULD clearly indicate the keyedReference that cannot be validated.
- **E_unvalidatable**: A keyedReference in a categoryBag that references a checked value set cannot be validated by the UDDI node because the referenced tModel has been marked unvalidatable. The error text SHOULD clearly indicate the keyedReference that cannot be validated.
- **E_valueNotAllowed**: The value set validation routine chosen by the UDDI node has rejected the businessService data provided. The error text SHOULD clearly indicate the keyedReference that was not successfully validated.

5.2.18 save_tModel

The save_tModel API call adds or updates one or more registered tModel elements. tModels MAY be signed and tModels MAY be saved with publisher-assigned keys, including those tModels that establish the domain partition of publisher-assigned keys, known as domain key generator tModels.

5.2.18.1 Syntax:



5.2.18.2 Arguments:

- ***authInfo***: This optional argument is an element that contains an authentication token. Authentication tokens are obtained using the get_authToken API call or through some other means external to this specification. Registries that serve multiple publishers and registries that restrict who can publish in them typically require authInfo for this call.
- ***tModel***: Required repeating element containing one or more required repeating complete tModel elements. For the purpose of performing round-trip updates, this data can be obtained in advance by using the get_tModel API call or by other means.

5.2.18.3 Behavior:

If the *uddiKey* value within a tModel (i.e., tModelKey) is missing or is passed with an empty value, this is a signal that a new tModel is being inserted and that the UDDI registry MUST assign a new tModelKey identifier to this data. If the new tModel is categorized with the *keyGenerator* value from the uddi:uddi.org:categorization:types category system, any publisher assigned key MUST end with the string "*:keygenerator",* making the tModel a key generator tModel. If the new tModel is categorized with the *keyGenerator* value from the uddi:uddi.org:categorization:types category, an empty uddiKey signifies that the tModelKey generated by the node will end with the string "*:keygenerator*", making the tModel a key generator tModel. New tModels can also be added with publisher-assigned keys. See Section 5.2.2.2 *Behavior of Publishers* and Section 5.2.18.3.1 *Domain Key Generator tModels*.

This API call performs an update to existing registered data when the tModelKey values have *uddiKey* values that correspond to already registered data.

If a tModelKey value is passed that corresponds to a tModel that was previously hidden via the delete_tModel API call, the save_tModel service restores the tModel to full visibility, making it available for return in find_tModel results.

The value of the deleted attribute in the tModel is set to false in all saves.

Multiple representations of the overview document MAY be registered for a tModel allowing, for example, both technical and human readable representations of the technical overview to be provided.

When a tModel is saved with keyedReferences, all tModelKeys used in keyedReferences must refer to tModels that existed prior to processing the tModel containing the references. A save_tModel API call may contain a sequence of tModels, in which case a keyedReference in a tModel may refer to tModelKeys created earlier but not later in the sequence. A tModel being created must not refer to itself. Self-referencing tModels can be created by using two subsequent save_tModel API calls, the first one without the reference, and the second one with the reference (to the already saved tModel). If these conditions are not met, the node MUST return E_invalidKeyPassed.

When a tModel is saved with identifierBag or categoryBag contents that is associated with a checked value set or category group system tModel, the references MUST be checked for validity prior to completion of the save, or the node MUST return E_unsupported, indicating it does not support the referenced checked value set or category group system. See Section 5.2.3 *Special considerations for validated value sets,* Appendix E *Using Identifiers* and Appendix F *Using Categorization* for additional details.

5.2.18.3.1 Domain key generator tModels

For registries that use the recommended key syntax, a domain key generator tModel establishes a key partition from which *uddiKey*s can be derived and used in other entities controlled by the publisher, as described in Section 4.4.1 *Key Syntax*. Additional considerations are involved when publishing a domain key generator tModel for the first time.

1. The tModelKey MUST be in the form of a *domain_key* and MUST end with the term: *keyGenerator*.
2. The tModelKey MUST be categorized with the *keyGenerator* value from the uddi:uddi.org:categorization:types category system.
3. Registry policy for establishing key domains MAY require the tModel to be signed.

Also, publishers of key generator tModels MAY use the overviewDoc to describe how the key space is defined.

The save_tModel API call does a first pass check of the tModel to check its suitability and, if it is acceptable according to the policy of the registry for saving domain key generator tModels, returns the tModelDetail for the registry. If it is not acceptable the reason is clearly indicated in the returned dispositionReport and no further processing takes place.

If the registry has multiple nodes, returning the tModelDetail is not an indication that the domain key generator tModel has been published successfully. A registry that allows publisher assigned keys MUST have a policy to ensure domainKey collisions do not occur. The custodial node MUST ensure that the domain key generator tModel is not in the process of being published simultaneously on some other node. If, after the conclusion of a full replication cycle, no UDDI node has already assigned or attempted to assign the partition (e.g., no change record has been received from other nodes), the custodial node completes the publish operation of the domain key generator tModel, assigning it to the publisher. If some other node has already been assigned the partition, the tModel is not published. See Section 7.3.9 changeRecordNewDataConditional for more information on the replication structure, and Section 9.4.2 General Keying Policy and Section 9.4.3 Policy Abstractions for the UDDI keying scheme for the recommended policy that addresses acceptance of a domain key generator.

When the publishing of a domain key generator tModel has completed, the custodial node MAY notify the publisher that the tModel is ready for use. Whether a node does this and the means by which it does so is a node policy. A typical node policy is to notify the publisher by e-mail using an e-mail address gathered at the time the publisher account was set up.

Before the publish operation is complete, the domain key generator tModel will be ignored by find_xx and get_xx API calls, and will return an E_keyUnavailable error to further save_tModel calls.

If after the replication cycle the publisher is in doubt about the outcome, get_tModelDetail may be issued specifying the key of the domain key generator tModel being published. If a tModel is retrieved and the publisher is the owner, the operation succeeded. If a tModel is retrieved and some other publisher is the owner, the operation failed because another publisher published a domain key generator with the chosen *domain_key* first. If no tModel is retrieved, then either the registry experienced a failure, or two publishers tried to publish tModels with the same key "simultaneously", and neither succeeded. In either of these cases, the save_tModel operation may be retried.

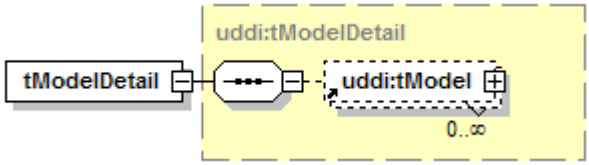
Attempts to remove the following categorization from a successfully published key generator tModel will fail with E_fatalError, since it is this very categorization that distinguishes key generator tModels from other tModels:

```
<tModelKey="uddi:uddi.org:categorization:types" keyValue="keyGenerator" />
```

5.2.18.4 Returns:

In most cases this API returns a tModelDetail containing the final results of the call that reflects the new or pending registered information for the affected tModel structures. Any tModelKey attributes that were assigned as a result of processing the save_tModel API are included in the tModel data. When a domain key generator is saved for the first time, the tModel that is returned in the tModelDetail represents an interim state, until all nodes in the registry have ascertained that the requested key domain does in fact belong to the publisher[27]. See Section 7.3.9 *changeRecordNewDataConditional* for more information. If multiple tModel elements are passed in the save_tModel request, the order of the response MUST exactly match the order that the elements appeared in the save. If the same tModel, determined by matching key, is listed more than once in the save_tModel API, it MAY be listed only once in the result for each appearance in the save_tModel API. If the same tModel appears more than once in the response, the last occurrence of the tModel in the results represents the state stored in the registry.

The tModelDetail has the form:



5.2.18.5 Caveats:

If an error occurs in processing this API call, a dispositionReport element MUST be returned to the caller in a SOAP Fault. See Section 4.8 *Success and Error Reporting.* In addition to the errors common to all APIs, the following error information is relevant here:

- **E_accountLimitExceeded**: Signifies that user account limits have been exceeded.
- **E_invalidKeyPassed**: Signifies that the request cannot be satisfied because one or more *uddiKey* values specified are not valid key values for the entities being published. tModelKey values that either do not exist, or exist with a different entity

[27] It is not possible to deterministically know when a save_tModel API call for a keyGenerator tModel has been unsuccessful when such failure occurs as a result of checking performed subsequent to the synchronous response provided with the save_tModel API. Nodes are, however, encouraged to provide publishers with details on such failures using out-of-band communication mechanisms, such as e-mail

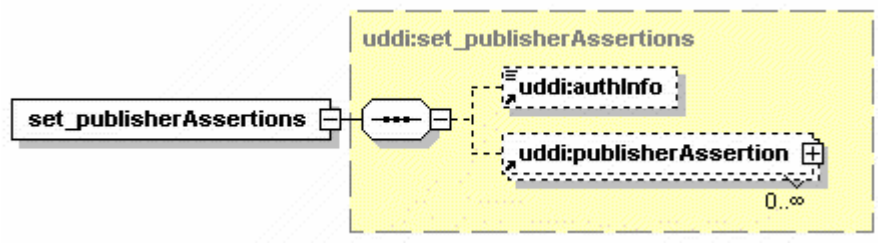
type, or are not authorized to be proposed by the publisher are considered to be invalid values. The key causing the error SHOULD be clearly indicated in the error text.

- **E_invalidValue**: A value that was passed in a keyValue attribute did not pass validation. This applies to checked value sets referenced using keyedReferences. The error text SHOULD clearly indicate the key and value combination that failed validation.
- **E_keyUnavailable**: Indicates that the proposed key has already been assigned to some other publisher, is not within the partition defined by a key generator tModel that the publisher owns, or, in the case of a domain key generator tModel being saved for the first time, is assigned to some other publisher or is still pending its first save.
- **E_requestTimeout**: Signifies that the request could not be carried out because a needed validate_values Web service did not respond in a reasonable amount of time. Details identifying the failing Web service SHOULD be included in the dispositionReport element.
- **E_unacceptableSignature**: Indicates that the digital signature in the tModel is missing or does not meet the requirements of the registry. The errInfo element provides additional details.
- **E_userMismatch**: Signifies that one or more of the uddiKey values passed refers to data that is not owned by the individual publisher who is represented by the authentication token.
- **E_unsupported**: A keyedReference in a categoryBag or an identifierBag that references a checked value set cannot be validated by the UDDI node because the node does not support the referenced checked value set. The error text SHOULD clearly indicate the keyedReference that cannot be validated.
- **E_unvalidatable**: A keyedReference in a categoryBag or an identifierBag that references a checked value set cannot be validated by the UDDI node because the referenced tModel has been marked unvalidatable. The error text SHOULD clearly indicate the keyedReference that cannot be validated.
- **E_valueNotAllowed**: Restrictions have been placed by the value set provider on the types of information that should be included at that location within a specific value set. The validation routine chosen by the UDDI node has rejected this tModel for at least one specified keyedReference. The error text SHOULD clearly indicate the keyedReference that was not successfully validated.

5.2.19 set_publisherAssertions

The set_publisherAssertions API call is used to manage all of the tracked relationship assertions associated with an individual publisher. See Appendix A *Relationships and Publisher Assertions* for more information.

5.2.19.1 Syntax:



5.2.19.2 Arguments:

- ***authInfo***: This optional argument is an element that contains an authentication token. Authentication tokens are obtained using the get_authToken API call or through some other means external to this specification. Registries that serve multiple publishers and registries that restrict who can publish in them typically require authInfo for this call.
- ***publisherAssertion***: Optional repeating element asserting a relationship. Relationship assertions consist of a reference to two businessEntity key values as designated by the fromKey and toKey elements, as well as a REQUIRED expression of the directional relationship within the contained keyedReference element. See Appendix A *Relationships and Publisher Assertions*. The fromKey, the toKey, and all three parts of the keyedReference – the tModelKey, the keyName, and the keyValue – MUST be specified. E_fatalError is returned if any of these elements are missing in any of the publisherAssertion elements. Empty (zero length) keyNames and keyValues are permitted.

5.2.19.3 Behavior:

The full set of assertions associated with a publisher is effectively replaced whenever this API is used. When this API call is processed, the publisher assertions that exist prior to this API call for a given publisher are examined by the UDDI registry. Any new assertions not present prior to the call are added to the assertions attributed to the publisher. Any existing assertions not present in the call are deleted. As a result, new relationships may be completed (e.g. determined to have a completed status), and existing relationships may be dissolved. Invoking this API with no publisherAssertion elements deletes all assertions associated with the publisher.

Any relationships attributed to assertions previously present but not present in the data provided in this call are deactivated and are no longer visible via the find_relatedBusinesses API. For the sake of determining uniqueness within an assertion set, the fromKey, toKey, and the entire keyedReference within the publisherAssertion element are significant. Any differences in any of the individual publisherAssertion element contents constitute a new unique assertion for purposes of detecting new assertions. The direction of the relationship, as

indicated by the two businessKey values in the fromKey and toKey elements, is also relevant in determining assertion uniqueness.

The publisher must own the businessEntity referenced in the fromKey, the toKey, or both. If both of the businessKey values passed within an assertion are owned by the publisher, then the assertion is automatically complete and the relationship described in the assertion is visible via the find_relatedBusinesses API. To form a relationship when the publisher only owns one of the two keys passed, the assertion MUST be matched exactly by an assertion made by the publisher who owns the other business referenced. Assertions exactly match if and only if they:

1. refer to the same businessEntity in their fromKeys;
2. refer to the same businessEntity in their toKeys;
3. refer to the same tModel in their tModelKeys;
4. have identical keyNames; and
5. have identical keyValues.

When a publisherAssertion that is being saved references a checked relationship system using the tModelKey in the contained keyedReference, the reference MUST be checked for validity prior to completion of the save, or the node must return E_unsupported, indicating it does not support the referenced checked relationship system. Validation of a relationship system reference entails verification that the reference is valid according to the validation algorithm defined for the relationship system and described by its tModel. For cached checked relationship system, the validation algorithm verifies that referenced keyedReferences are valid for the relationship system.

For registries supporting the subscription APIs at any node, it is necessary to track a modified date for publisherAssertion elements so that nodes have the necessary information for responding to subscription requests involving find_relatedBusinesses and get_assertionStatusReport filters.

5.2.19.4 Returns:

Upon successful completion, a publisherAssertions structure is returned containing all of the relationship assertions currently attributed to the publisher. When registries distinguish between publishers, the structure contains assertion data that is associated with the authInfo passed.

See Section 5.2.13.3 *get_publisherAssertions* for more information on the publisherAssertions structure and contents.

This API returns all assertions made by the publisher who was authenticated in the set_publisherAssertions API.

5.2.19.5 Caveats:

If an error occurs in processing this API call, a dispositionReport element MUST be returned to the caller in a SOAP Fault. See Section 4.8 *Success and Error Reporting.* In addition to the errors common to all APIs, the following error information is relevant here:

- **E_invalidKeyPassed**: Signifies that one of the *uddiKey* values passed did not match with any known businessKey or tModelKey values. The assertion element and the key that caused the problem SHOULD be clearly indicated in the error text.
- **E_userMismatch**: Signifies that neither of the businessKey values passed in the embedded fromKey and toKey elements is controlled by the publisher associated with the authentication token. The error text SHOULD clearly indicate which assertion caused the error.

5.3 Security Policy API Set

The security API includes the following API calls:

- **discard_authToken**: Used to inform a node that a previously obtained authentication token is no longer required and should be considered invalid if used after this message is received.
- **get_authToken**: Used to request an authentication token in the form of an authInfo element from a UDDI node. An authInfo element MAY be required when using the API calls defined in Section 5.1 *Inquiry API Set*, Section 5.2 *Publication API Set*, Section 5.4 *Custody and Ownership Transfer API Set,* and Section 5.5 *Subscription API Set*.

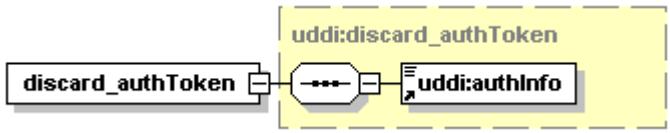
Whether authInfo elements are required on API calls is determined by node policy as described in Section 4.8 *About Access Control and the authInfo Element*. In the event that an authInfo element is not discarded, a node MAY choose to expire the authentication token so it is no longer valid for authentication in API calls after a period of time. If an expired token is passed to an API call other than discard_authToken, the error E_authTokenExpired will be returned as described in Chapter 12, *Error Codes*.

A UDDI node typically does not support the Security API set if it does not support using an authInfo element in any API set. If the node does support using an authInfo element in any of the API set provided by the node, it SHOULD support the Security API set. A node MAY provide an alternative mechanism for obtaining authInfo elements.

5.3.1 discard_authToken

The discard_authToken API call is used to inform a node that the passed authentication token is to be discarded, effectively ending the session.

5.3.1.1 Syntax:



5.3.1.2 Arguments:

- ***authInfo***: This required argument is an element that contains an authentication token. Authentication tokens are obtained using the get_authToken API call.

5.3.1.3 Behavior:

Discarding an expired authToken is processed and reported as a success condition.

5.3.1.4 Returns:

Upon successful completion, an empty message is returned. See section 4.8 *Success and Error Reporting*.

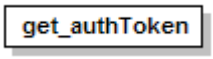
5.3.1.5 Caveats:

If an error occurs in processing this API call, a dispositionReport structure will be returned to the caller in a SOAP Fault.

5.3.2 get_authToken

The get_authToken API call is used to obtain an authentication token. An authToken element MAY be required when using the API calls defined in Section 5.1 *Inquiry API Set*, Section 5.2 *Publication API Set*, Section 5.4 *Custody and Ownership Transfer API Set,* and Section 5.5 *Subscription API Set*.

5.3.2.1 Syntax:



Attributes

Name	Use
userID	Required
cred	Required

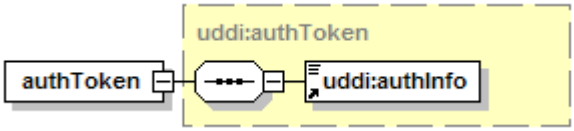
5.3.2.2 Arguments:

- ***userID***: This required attribute argument is the user identifier that an individual authorized user was assigned by a UDDI node. Nodes SHOULD provide a means for individuals to obtain a userID and password credentials that will be valid at the given node.
- ***cred***: This required attribute argument is the password or credential that is associated with the user.

5.3.2.3 Returns:

Upon successful completion this API call returns an authToken structure that contains a valid authInfo element that can be used in subsequent calls to API calls that require an authInfo value.

The authToken message has the form:



The authToken structure contains a single authInfo element that represents a token that is to be passed back in API calls that require one. This structure is always returned as a synchronous response to the get_authToken message.

5.3.2.4 Caveats:

If an error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP Fault. In addition to the errors common to all APIs, the following error information is relevant here:

- **E_unknownUser**: Signifies that the UDDI node that received the request does not accept the userID and/or cred argument values passed as valid credentials.

5.4 Custody and Ownership Transfer API Set

This section defines the UDDI Custody and Ownership Transfer API Set[28]. Data custody is introduced in Section 1.5.6 *Data Custody.* Ownership transfer is introduced in Section 1.5.7 *Transfer of Ownership.* By virtue of having created an entity, a publisher has ownership of the entity and is said to be the owner of the entity. A custodial node MUST maintain a relationship of ownership between an entity and its publisher by means of authorization mechanisms. Every node of a multi-node registry MUST guarantee the integrity of an entity's custody. As such, a node MUST not permit changes to an entity unless it has custody of it.

The Custody and Ownership Transfer API Set enables any nodes of a registry to cooperatively transfer custody of one or more businessEntity or tModel structures from one node to another, as well as allowing the transfer of ownership of these structures from one publisher to another. Associated entities of a businessEntity such as its businessService, bindingTemplate, and publisherAssertion structures are transferred as part of the custody transfer of the business entity.

From a custody transfer point of view, the publishers are always distinct, though it may be the case that the publishers are the same person. Also, the two nodes may or may not be distinct; intra-node transfer between two publishers is simply a degenerate case in which node custody does not change. Thus, in the case of an inter-node transfer, ownership transfer is implied. In the case of an intra-node transfer the behavior results in the transfer of ownership between two publishers.

For example, one UDDI registry, UDDI-1, MAY allow each node in UDDI-1 (composed of nodes 1A, 1B and 1C) to define its own policies for registration, authentication and authorization. In this case, a "person", (P1) would need to review the policies of all 3 nodes and decide upon the node with which it chooses to register with. P1 may choose to register with more than one node. P1 registers with node1A . Node1A also specifies how P1 is authenticated. If P1 successfully authenticates and publishes a business entity (BE1) then P1 becomes the "owner" of BE1. Node1A is said to be the "custodian" of BE1. P1 can also register at node1B. If P1 successfully authenticates and publishes a business entity (BE2) then P1 becomes the "owner" of BE2. Node1B is said to be the "custodian" of BE2. There is no assumption that the registry UDDI-1 or its nodes (node1A and node1B) are aware that P1 is the same "person". P1 is responsible for maintaining the appropriate identity and authenticating correctly to each node within a registry.

Another UDDI registry, UDDI-2, MAY require each of its nodes (node2-1, node2-2 and node2-3) to use the same registration, authentication and authorization mechanisms. In this case, the policies are the same across all nodes. The relationship of registration, publication and ownership remains the same. If P1 wants to register with different nodes in UDDI-2, then it needs to differentiate its registration with the different nodes, since an attempt to register at node2-2 after registering at node2-1, would fail as "already registered" (since by policy the nodes all share the same registration, authentication and authorization).

5.4.1 Overview

There are a number of scenarios where a publisher may choose to transfer custodianship or ownership of one or more entities. These are described in this section.

[28] The Custody Transfer tModel is but one of several tModels defined for UDDI. See Section 11.3.5 UDDI Custody Transfer API tModel for more information.

5.4.1.1 Intra-Node Ownership Transfer

Intra-node ownership transfer involves transferring entity ownership from one publisher to another within the same UDDI node. Usage scenarios for this type of transfer include the following:

- Businesses or organizational merges: Multiple organizations need to be consolidated under the control of a single publisher.
- Domain key generators: One use of ownership transfer is the transfer of ownership of a derived key generator from one publisher to another to enable her or him to publish entities using keys in that domain.

The save_xx APIs can also be used to move entities between parent entities that are owned by the same publisher. The save_service API, for example, can be used to move services (and binding templates) between one business entity and another as described in Section 5.2.17.3 *Behavior* of the save_service API. Changing the parent relationship in this way causes two businessEntity structures to be changed. Doing so enables the following scenarios:

- Divestitures: An organization needs to reassign the control of a set of services to two or more publishers.
- Consolidation of registry entities: There are multiple entities for a given business that are to be consolidated under a single publisher.

5.4.1.2 Inter-Node Custody Transfer

Inter-node custody transfer involves the custody transfer of a set of entities across nodes of a UDDI registry. A transfer of ownership ensues as a consequence of this custody transfer. In addition to the intra-node scenarios described above, inter-node custody transfer may be used to address the following use cases:

- Unsatisfactory service level: The functionality or service level provided by a given node operator is insufficient, and the publisher wishes to move their UDDI data to another node.
- Change in availability for a UDDI node: A node is no longer providing UDDI services, and all publishers need to be migrated to one or more nodes of the registry.
- Organizational Mergers, Divestitures or Consolidations: Changes in organizational structure may result in the need to make changes to the set of publishers used to manage the entities at various nodes of a registry.

For any of these intra and inter-node scenarios, a mechanism is specified to facilitate the transfer the custody of businessEntity and tModel entities between nodes whether the entity is being transferred within a single node or whether a custody transfer occurs between nodes of a registry.

5.4.2 Custody Transfer Considerations

When a businessEntity is transferred, all related businessService and bindingTemplate elements are transferred as well. In addition, any publisherAssertion elements that reference the businessEntity element's businessKey that are owned by the publisher are also transferred.

Note that the relinquishing publisher is not required to transfer all of its UDDI entities (i.e. businessEntity and/or tModel entities) in a single custody transfer request, nor is it required to transfer all of its entities to the same target publisher or target node. Any combination or subset of UDDI registry entities may be transferred to any number of target publishers or nodes.

5.4.3 Transfer Execution

The Custody and Ownership Transfer API Set enables two publishers P1 and P2 and two nodes, N1 and N2, in a registry to cooperatively transfer custody of one or more existing businessEntity or tModel structures, E1…En, from N1 to N2 and, and by extension to transfer ownership of the entities from P1 to P2. Related businessService, bindingTemplate, and publisherAssertion structures are transferred with their related businessEntities. From the registry's point of view, the publishers are always distinct, though it may be the case that P1 and P2 are the same party. The two nodes may or may not be distinct; intra-node transfer of ownership from P1 to P2 is simply a degenerate case in which node custody does not change.

The Custody and Ownership Transfer API Set is divided into two parts, a set of two client APIs and a single inter-node API. These client APIs are get_transferToken and transfer_entities; in short, this constitutes the Ownership Transfer API portion of this API set. The inter-node-API is transfer_ custody which when combined with replication makes up the Custody Transfer API portion of this API set.

The overall flow of custody and ownership transfer is as follows:

Publisher P1 invokes get_transferToken on N1, specifying the keys K1…Kn of the entities E1…En that are to be transferred. If P1 is authorized to do this (i.e., if P1 has ownership of E1…En), N1 returns a structure T, called a transfer token, that represents authority to transfer the entities, including all of the naturally contained children and publisher assertions related to business entities involved in the transfer that are owned by P1. The transferToken is a structure that consists of an opaque string that is meaningful only to the node that issued it, an expiration time, and a node identifier.

P1 then gives T to P2 (typically by some secure means since T is valuable). The publisher obtaining the custody information needs to have previously obtained a publishers account on the node accepting custody of the entity before he/she can complete the custody transfer. P2 then invokes transfer_entities on N2, passing K1…Kn and T. If transfer_entities completes successfully, the entities E1…En and their related structures (businessService, bindingTemplate, and publisherAssertion) are in the custody of N2 and are owned by P2. If the operation fails, nothing happens to the entities. The actual transfer proceeds as follows, in the processing of transfer_entities.

If N1 and N2 are not distinct nodes, the ownership transfer from P1 to P2 is an operation that is purely internal to the node – how it happens is up to the implementation. If N1 and N2 are distinct, the following protocol occurs while processing the transfer_entities request on N2.

Upon receipt of a transfer_entities request, N2 checks that K1…Kn are valid keys. There is the possibility that P1 might transfer more data than P2 can accept due to policy-based restrictions on the limit of entities allowed to be owned by P2 at N2. As is described below, replication is used to complete the custody transfer process. A question that arises is at the time of accepting the datum related to the transfer, could N2 throw a replication error because the data being transferred exceeds the limits of user P2? Such limits can not be enforced during replication because they are node-local policy decisions from the perspective of enforcement. Thus, it is therefore possible that as a result of a custody transfer a publisher may be caused to hold more data that he/she would have been able to publish. Should this situation occur, P2 MUST not be allowed to publish any additional data unless P2 first reduces the number of entries it owns to an allowable limit.

If all is well, N2 invokes the inter-node API transfer_custody on N1, presenting the keys of top-level entities to be transferred, K1…Kn, P2's identity (using the publisher's authorizedName), N2's node identifier (as known in the Replication Configuration structure, see Section 7.5.2 *Configuration of a UDDI Node – operator element*), and T. The transferToken, T, implies permission to transfer the entire content of the entities it identifies, including all of the contained entities and related publisherAssertions, if any. N1 checks to see whether T is a valid transferToken that it issued and that T represents the authority to transfer E1…En. If the validation is successful, N1 prevents further changes to entities E1…En. N1 then updates the

authorizedName and nodeID of the operationalInfo of E1…En and related entities so that they are shown to be in the custody of N2 and owned by P2. Finally, N1 responds to N2 which triggers N2 to respond to the transfer_entities caller. This completes the processing for the transfer_entities request.

In the case that the datum being transferred is a key generator tModel, N1 will disallow further generation of keys associated with this key partition at its node.

Following the issue of the empty message by N1 to the transfer_custody call, N1 will submit into the replication stream a changeRecordNewData providing in the operationalInfo, N2's nodeID identifying it as the node where the datum is being transferred to, and the authorizedName of P2. The acknowledgmentRequested attribute of this change record MUST be set to "true".

The last modified date timestamp in the operationalInfo must change to reflect the custody transfer. Figure 2 depicts the flow of a custody transfer between P1 and P2.

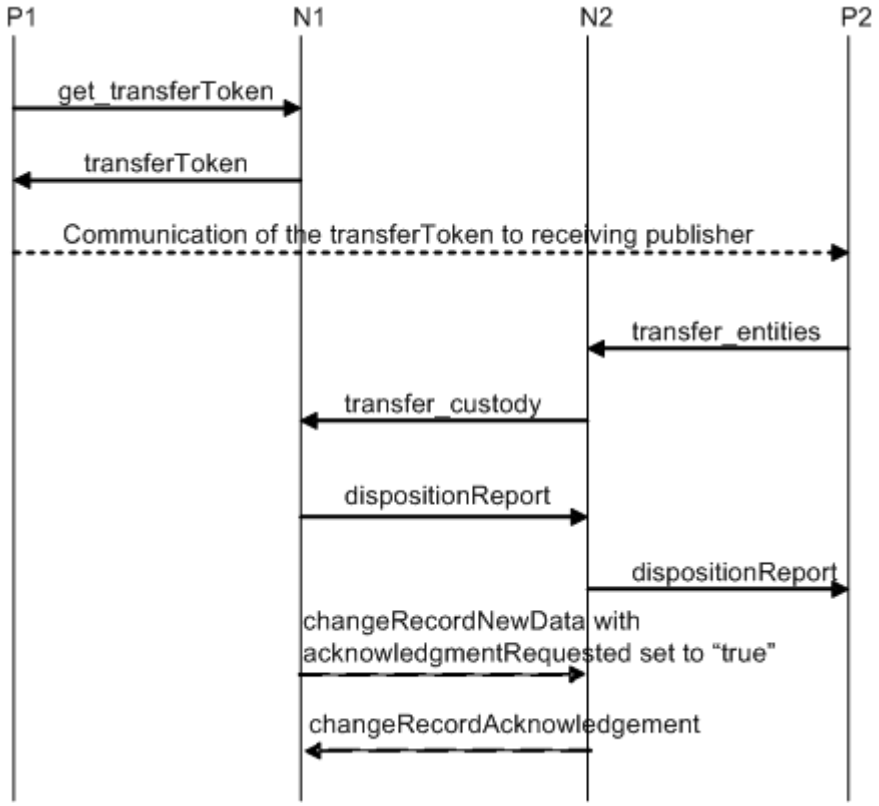


Figure 2 - Custody Transfer

Once N2 receives the changes via the replication stream it assumes custody of E1…En and assigns ownership of the entities and related entities owned by P1 to P2.

5.4.3.1 Content of a transferToken

The transferToken is a structure that consists of an opaque string that is meaningful only to the node that issued it, an expiration time, and a node identifier. It represents the one-time authority to transfer ownership of a specific set of entities to any publisher and to transfer

custody of them to any node in the registry. Issuing it does not cause any such transfer to occur; it is simply an authorization for such a transfer to take place. The authority represented by a transferToken expires after some period of time, per node policy.

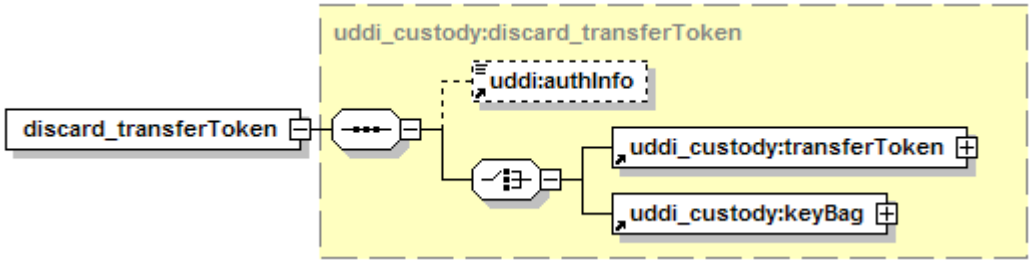
The expiration time represents the time, according to the node that issued the transferToken, after which the token is no longer valid. The expiration time is only a convenience; changing it does not change the time at which the authority expires. The node identification identifies, in a registry dependent way, the node that issued the transferToken. It is used by the node that receives the transferToken to determine which node in the registry to direct the transfer_custody request to.

The opaque token SHOULD reflect the entities that the custodial publisher has sought permission to be transferred. It is often used in the latter stages of the custody transfer process by the custodial node to verify that the entities that the target publisher is requesting to own have been authorized by the target node under the auspices of the transferToken.

5.4.4 discard_transferToken

The discard_transferToken API is a client API used to discard a transferToken obtained through the get_transferToken API at the same node. This API accepts either a transferToken or a keyBag as parameters to remove the permission to transfer data associated with a particular transferToken. If a keyBag is provided, all tokens corresponding to the keys in the keyBag will be discarded and will no longer be valid for custody or ownership transfer after the discard_transferToken is processed, irrespective of whether the keys match any known business or tmodelKey values. In the event that the keyBag represents a subset of the keyBag for one or more transferToken elements, the transferToken is discarded and will no longer be valid for transferring any entity. If the token passed in the transferToken argument does not match an existing token known to the system, no action is taken and success is reported. Keys in the keyBag argument that do not have a corresponding token are ignored.

5.4.4.1 Syntax



5.4.4.2 Arguments

- ***authInfo***: This OPTIONAL argument is an element that contains an authentication token. Authentication tokens are obtained using the get_authToken API call or through some other means external to this specification, and represent the identity of the publisher at a UDDI node.
- ***transferToken***: This is a known transferToken obtained by a publisher at the node where the get_transferToken API was invoked.
- ***keyBag***: One or more uddiKeys associated either with businessEntity or tModel entities owned by the publisher that were to be transferred to some other publisher and/or node in the registry as the result of invocation of get_transferToken. At least one businessKey or tModelKey must be provided in a keyBag.

5.4.4.3 Returns

Upon successful completion, an empty message is returned. See section 4.8 *Success and Error Reporting.*

No error will be reported if the transferToken provided in the call does not match an existing token. No error will be reported if a token is not found for a particular key in the keyBag.

5.4.4.4 Caveats

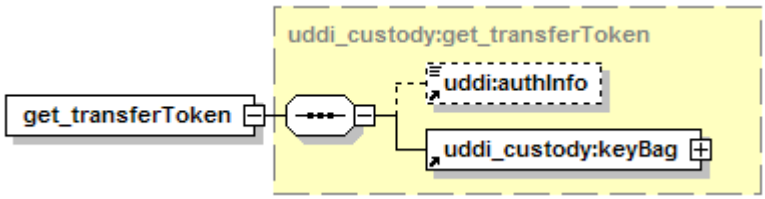
If an error occurs in processing this API call, a dispositionReport structure MUST be returned to the caller in a SOAP Fault. See Section 4.8 *Success and Error Reporting.* In addition to the errors common to all APIs, the following error information is relevant here:

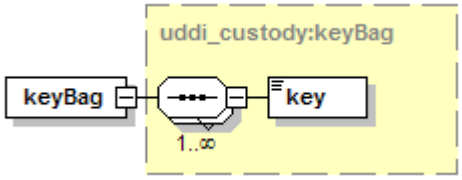
- **E_invalidKeyPassed**: signifies that one of the *uddiKey* values passed for entities to be transferred did not match with any known businessKey or tModelKey values. The key and element or attribute that caused the problem SHOULD be clearly indicated in the error text.

5.4.5 get_transferToken

The get_transferToken API is a client API used to initiate the transfer of custody of one or more businessEntity or tModel entities from one node to another. As previously stated, the two nodes may or may not be distinct; intra-node transfer between two publishers is simply a degenerate case in which node custody does not change. No actual transfer takes place with the invocation of this API. Instead, this API obtains permission from the custodial node, in the form of a transferToken, to perform the transfer. The publisher who will be recipient of the transferToken returned by this API must invoke the transfer_entities API on the target custodial node to actually transfer the entities.

5.4.5.1 Syntax





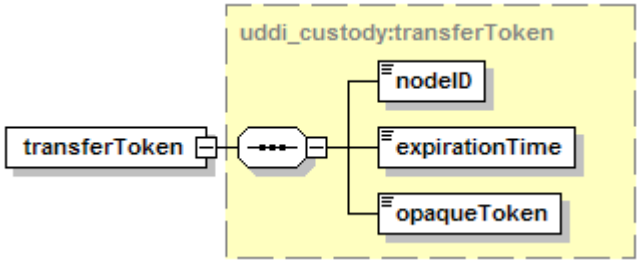
5.4.5.2 Arguments

- ***authInfo***: This OPTIONAL argument is an element that contains an authentication token. Authentication tokens are obtained using the get_authToken API call or through some other means external to this specification and represent the identity of the publisher at a UDDI node.
- ***keyBag***: One or more key (of type uddi:uddiKey) associated either with businessEntity or tModel entities owned by the publisher that are to be transferred to

some other publisher and/or node in the registry. At least one businessKey or tModelKey must be provided.

5.4.5.3 Returns

If the publisher identified by the *authInfo* element owns the businessEntity elements identified by the *businessKey* elements provided and the tModel elements identified by the *tModelKey* elements provided, a transferToken is returned that represents the one-time permission to transfer custody of the identified entities.



The transfer token consists of a *nodeID,* an *expirationTime* and an *opaqueToken*. The *nodeID* is used during the transfer_entities API by the recipient node to confirm with the relinquishing custodial node that the custody transfer is authorized and still valid. The *nodeID* of the transferToken is the value of the *nodeID* element of the Replication Configuration Structure. Refer to Section 7.5.2 *Configuration of a UDDI Node – operator Element*.

The *expirationTime*, defined as xsd:dateTime, represents the time at which the transfer token is no longer valid.

The *opaqueToken* is only meaningful to the node that issues it. The *opaqueToken* is defined *as xsd:base64Binary* to allow for a RECOMMENDED encryption of the token under the relinquishing custody node's own encryption key.

5.4.5.4 Caveats

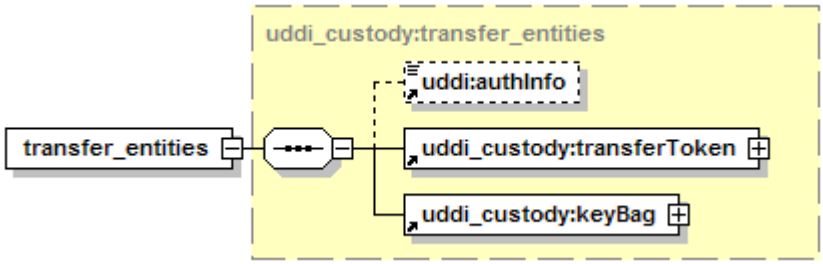
If an error occurs in processing this API call, a dispositionReport structure MUST be returned to the caller in a SOAP Fault. See section 4.8 *Success and Error Reporting.* In addition to the errors common to all APIs, the following error information is relevant here:

- **E_invalidKeyPassed**: signifies that one of the *uddiKey* values passed for entities to be transferred did not match with any known businessKey or tModelKey values. The key and element or attribute that caused the problem SHOULD be clearly indicated in the error text.
- **E_tokenAlreadyExists**: signifies that one or more of the businessKey or tModelKey elements that identify entities to be transferred are associated with a transferToken that is still valid and has not been discarded, used or expired. The error text SHOULD clearly indicate which entity keys caused the error.
- **E_userMismatch**: signifies that one or more of the businessKey or tModelKey elements that identify entities to be transferred are not owned by the publisher identified by the *authInfo* element. The error text SHOULD clearly indicate which entity keys caused the error.

5.4.6 transfer_entities

The transfer_entities API is used by publishers to whom custody is being transferred to actually perform the transfer. The recipient publisher must have an unexpired transferToken that was issued by the custodial node for the entities being transferred.

5.4.6.1 Syntax



5.4.6.2 Arguments

- ***authInfo***: This OPTIONAL argument is an element that contains an authentication token. Authentication tokens are obtained using the get_authToken API call or through some other means external to this specification, and represent the identity of the publisher at a UDDI node, in this case, the new owner of the entities being transferred.
- ***transferToken***: Required argument obtained from the custodial node via a call to get_transferToken by the publisher requesting a transfer of custody. The transferToken contains an opaque token, an expiration date, and the identity of the custodial node. The transferToken represents permission to transfer the entities that have been identified via a prior call to the get_transferToken API.
- ***keyBag***: One or more uddiKeys associated with businessEntity or tModel entities that are to be transferred to this publisher at the target node in the registry. The set of keys must be the same as the set of keys in the keyBag of the get_transferToken API call from which the given transferToken was once obtained.

5.4.6.3 Returns

The target node responds to this API by performing the transfer operation. This operation is comprised of four steps:

- Verification that the entity keys are valid.
- Verification that ownership of the entities by the recipient publisher is allowed and would not violate any policies at the target node related to publisher limits.
- Verification with the custodial node that the transfer of the designated entities is allowed. This is accomplished by invoking transfer_custody on the custodial node that is identified by the nodeID element in the transferToken. Any errors returned by the custodial node cause this API to fail and are propagated to the caller.
- Changing custody and ownership of the designated entities and entering these changes into the replication stream.

Upon successful completion, an empty message is returned indicating the success of the transfer operation. In the case of an inter-node custody transfer, while the transfer is in process, the entities being transferred are not available for modification. To determine the state of the data, UDDI clients can use the get_operationalInfo API to determine when custody and

ownership transfer has taken place. A change in the nodeID of the operationalInfo provides such an indication.

5.4.6.4 Caveats

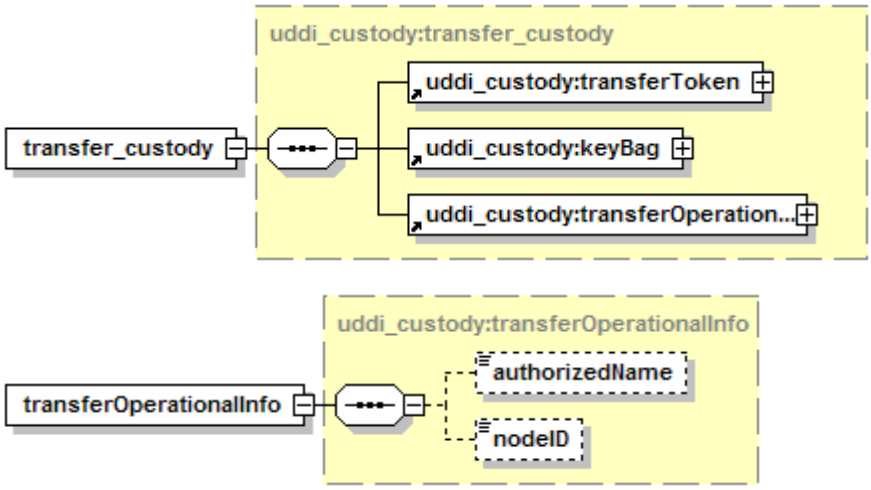
If an error occurs in processing this API call, a dispositionReport structure MUST be returned to the caller in a SOAP Fault. See Section 4.8 *Success and Error Reporting.* In addition to the errors common to all APIs, the following error information is relevant here:

- **E_accountLimitExceeded**: signifies that the target node has determined that the transfer of custody of the identified entities would result in the target publisher exceeding policy limits for the number of owned entities. The error text SHOULD clearly indicate which entities cause the publishers limits to be exceeded. It is possible for a publisher to come into possession of more data than the target node's policy allows. The condition and node behavior under these circumstances are described in Section 5.4.3*Transfer Execution.*
- **E_invalidKeyPassed**: signifies that one of the *uddiKey* values passed for entities to be transferred did not match with any known businessKey or tModelKey values. The key and element or attribute that caused the problem SHOULD be clearly indicated in the error text.
- **E_transferNotAllowed**: signifies that the transfer of one or more entities has been rejected by the target node or the custodial node. Reasons for rejection include expiration of the transferToken, use of an invalid transferToken, and attempts to transfer a set of entities that does not match the one represented by the transferToken. The reason for rejecting the custody transfer SHOULD be clearly indicated in the error text.

5.4.7 transfer_custody

Invoked by the target node in a custody transfer operation in response to transfer_entities, this API is used by the custodial node to ensure that permission has been granted to transfer custody of the entities that the target publisher has requested. The transfer_custody API is in the replication namespace since it is sent from one node to another node in a registry using replication.

5.4.7.1 Syntax



5.4.7.2 Arguments

- ***transferToken***: Required argument obtained from the custodial node via a call to get_transferToken by the publisher requesting a transfer of custody. The transferToken contains an opaque token, an expiration date, and the identity of the custodial node. The transferToken represents permission to transfer the entities that have been identified via a prior call to the get_transferToken API. The custodial node MUST verify that the transferToken has not expired and that the businessKey and tModelKey elements that the target publisher has provided in transfer_entities are allowed to be transferred as captured in the transfer token's opaqueToken.
- ***keyBag***: One or more uddiKeys associated with businessEntity or tModel entities that the target publisher is requesting ownership of at the target node in the registry. The set of keys must be the same as the set of keys in the keyBag of the get_transferToken API call from which the given transferToken was once obtained.
- ***transferOperationalInfo***: Required argument. The accepting publisher's authorizedName and the accepting node's nodeID are provided on input to the relinquishing custodial node to allow it to update the operationalInfo associated with the entities whose custody is being transferred. The authorizedName and nodeID elements are both required. The accepting node's nodeID is obtained via the Replication Configuration structure as described in Section 7.5.2 *Configuration of a UDDI Node – operator element*. The authorizedName is obtained from the call to transfer_entities by the requesting publisher.

5.4.7.3 Returns

The custodial node must verify that it has granted permission to transfer the entities identified and that this permission is still valid. This operation is comprised of two steps:

1. Verification that the transferToken was issued by it, that it has not expired, that it represents the authority to transfer no more and no less than those entities identified by the businessKey and tModelKey elements and that all these entities are still valid and not yet transferred. The transferToken is invalidated if any of these conditions are not met.
2. If the conditions above are met, the custodial node will prevent any further changes to the entities identified by the businessKey and tModelKey elements identified. The entity will remain in this state until the replication stream indicates it has been successfully processed via the replication stream.

Upon successful verification of the custody transfer request by the custodial node, an empty message is returned by it indicating the success of the request and acknowledging the custody transfer.

Following the issue of the empty message, the custodial node will submit into the replication stream a changeRecordNewData providing in the operationalInfo, the nodeID accepting custody of the datum and the authorizedName of the publisher accepting ownership. The acknowledgmentRequested attribute of this change record MUST be set to "true".

Finally, the custodial node invalidates the transferToken in order to prevent additional calls of the transfer_entities API.

5.4.7.4 Caveats

If an error occurs in processing this API call, a dispositionReport structure MUST be returned to the caller in a SOAP Fault. See Section 4.8 *Success and Error Reporting.* In addition to the errors common to all APIs, the following error information is relevant here:

- **E_transferNotAllowed**: signifies that the transfer of one or more entities has been rejected by the custodial node. Reasons for rejection include expiration of the transferToken and attempts to transfer a set of entities that does not match the one represented by the transferToken. The reason for rejecting the custody transfer SHOULD be clearly indicated in the error text.
- **E_invalidKeyPassed**: signifies that one of the *uddiKey* values passed for entities to be transferred did not match with any known businessKey or tModelKey values. The key and element or attribute that caused the problem SHOULD be clearly indicated in the error text.

5.4.8 Security Configuration for transfer_custody

The use of mutual authentication of UDDI nodes in conjunction with the transfer_custody API is RECOMMENDED. This MAY be achieved using mutual X.509v3 certificate-based authentication as described in the Secure Sockets Layer (SSL) 3.0 protocol. SSL 3.0 with mutual authentication is represented by the tModel uddi-org:mutualAuthenticatedSSL3 as described within Section 11.3.2 *Secure Sockets Layer Version 3 with Mutual Authentication*.

5.5 Subscription API Set

Subscription provides clients, known as subscribers, with the ability to register their interest in receiving information concerning changes made in a UDDI registry. These changes can be scoped based on preferences provided with the request. The APIs described below support this capability. Usage scenarios and examples are provided in Appendix C *Supporting Subscribers*. Each of the subscription APIs described here are OPTIONAL for UDDI implementations and MAY be implemented entirely at the discretion of a Node.

5.5.1 About UDDI Subscription API functions

The subscription API set satisfies a variety of requirements. The flexibility of the subscription API allows monitoring of activity in a registry by registering to track new, changed and deleted entries for each of these entities:

- businessEntity
- businessService
- bindingTemplate
- tModel
- related businessEntity
- publisherAssertion (limited to those publisherAssertions for which the subscriber owns at least one of the businesses referenced)

With the exception of single publisher registries subscription typically is limited to authorized clients as a matter of node policy. Therefore, subscribers MUST typically authenticate with the node before saving subscription requests. Individual nodes, including those in the UDDI Business Registry, MAY establish policies concerning the use of the subscription APIs they choose to offer. Such policies might include restricting the use of subscription, defining which APIs are supported, establishing whether subscriptions require authentication, defining special rules affecting different classes of subscriptions, or even imposing fees for the use of these services. The use of the authInfo argument is OPTIONAL throughout the subscription APIs, although registries which support multiple users or which require authentication for publishing operations typically require it.

Subscription allows subscribers to "monitor" a particular subset of data within a registry. Two patterns are defined. Nodes MAY support either or both:

- Asynchronous notification – subscribers choose to be asynchronously notified by the node when registry data of interest changes via calls to the notify_subscriptionListener API, which they implement as a "subscription listener" service.
- Synchronous change tracking – subscribers issue a synchronous request using the get_subscriptionResults API to obtain information on activity in the registry which matches their subscription preferences.

A subscription request establishes criteria for the subscription and specifies how and if the subscriber is to be notified of changes matching the specified criteria. Any of the existing standard inquiry APIs (find_xx and get_xx) may be used within a subscription request to define the criteria, although nodes are free to restrict which inquiry APIs are supported in subscription as a matter of policy. The duration, or life of a subscription is also a matter of node policy, but subscribers can renew existing subscriptions periodically instead of having to create new ones. Subscribers may also create multiple subscriptions. Each subscription request is treated independently. The level of detail provided for the data returned is controlled by the subscription request.

When asynchronous notifications are requested, subscriptions provide information on new, changed or deleted entities within a registry that occur *after* the point in time that the subscription is registered. A node notifies subscribers based upon its identification of data matching the requested subscription criteria. Subscribers can choose to have these notifications provided via email or an HTTP/SOAP-based Web service, which the subscriber MAY implement. Such services are called "subscription listeners." Notifications are made periodically rather than in response to the continuous stream of changes that normally occur within the registry. This means that subscription results provided via notifications pertain only to the *current* state of the entities at the time they are reported – intermediate state changes are not provided. While subscribers can specify a frequency for these notifications, nodes MAY choose to restrict this as a matter of policy.

When synchronous requests are made for subscription results, the *current* state of the registry data, which matches the subscription criteria, is returned for entries that were last created, changed or deleted within a specified date range. Prior states of the registry data are not available and are not returned.

Subscriptions are owned by the subscriber who creates them. A subscriptionKey, which distinguishes each individual subscription, is not visible to anyone except the subscriber. While node policy MAY permit others besides the subscription's owner to receive or retrieve subscription results, such interested parties require knowledge of the relevant subscriptionKey from the subscription owner in order to do so.

5.5.1.1 Definition of Changed Entities

As stated above, the subscription API allows monitoring of new, changed and deleted entities. This section provides the definition of changed entities. The following are the criteria for considering an entity to have been "changed":

- For businessEntity, businessService, bindingTemplate, and tModel:
  The entity is considered to be changed if the modifiedIncludingChildren element of the operationalInfo element of the entity has been changed.
- For publisherAssertion:
  A publisherAssertion is considered to be changed if the publisher has updated the publisherAssertion via the set_publisherAssertions, or add_publisherAssertions APIs.
- For related businessEntity:
  A related businessEntity (related to the business specified in the businessKey argument of find_relatedBusinesses API) is considered to be changed if either:
  1. the related businessEntity is changed, or
  2. at least one of the two reciprocal publisherAssertions that represents the relationship is changed.

5.5.2 Specifying Durations

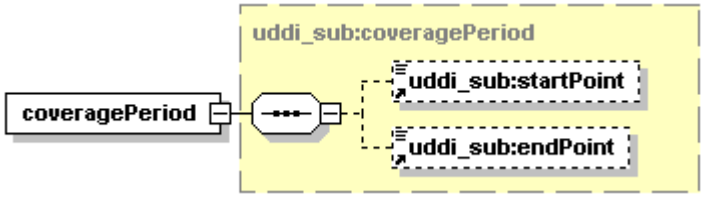
Time durations used in the subscription APIs are of type xsd:duration defined in XML Schema from [ISO 8601]. Any form supported by this data type is permitted. For example, the lexical representation extended format can be used, which is of the form PnYnMnDTnHnMnS, where nY represents the number of years, nM the number of months, nD the number of days, 'T' is the date/time separator, nH the number of hours, nM the number of minutes and nS the number of seconds. The "P" identifies the field as duration. The number of seconds can include decimal digits to arbitrary precision.

5.5.3 Specifying Points in Time

Points in time used in the subscription APIs are all of the XML Schema type, xsd:dateTime. Two points in time are used to specify a period of time.

5.5.4 Subscription Coverage Period

The APIs, which support each of the patterns previously described for obtaining subscription results, accept a coveragePeriod argument, which is composed of two points in time. Each of these corresponds to the last point in time which any given entity in the registry was modified (i.e., created, deleted or changed). The syntax of this element is:



Where*:*

- **startPoint**: Signifies the point in time after which subscription results are to be collected and/or returned. The startPoint is optional. If it is not specified, this indicates that all available results are to be returned from the beginning of the registry.
- **endPoint**: Signifies the point in time corresponding to the last activity date for entities matching subscription results. No activity among matching entities, which occurred after this point in time, is returned. The endPoint is optional. If not provided, it signifies that the latest changes available, which match the subscription criterions that are to be returned.

With respect to notifications, the startPoint of a given notification SHOULD align with the endPoint of the previous notification. If this is not the case, the subscriber SHOULD assume a notification was missed, or lost. The subscriber can then take corrective action by using the get_subscriptionResults API. Note that it is permissible for nodes to send the same data more than once, depending on overlaps in these times.

5.5.5 Chunking of Returned Subscription Data

If a subscriber specifies a maximum number of entries to be returned with a subscription and the amount of data to be returned exceeds this limit, or if the node determines based on its policy that there are too many entries to be returned in a single group, then the node SHOULD provide a chunkToken with results. The chunkToken is a string based token which is used by the node to maintain the state of the subscription results for a particular caller, when these results are chunked across multiple responses. The format and content of the chunkToken is a matter of implementation choice by individual nodes. The chunkToken returned with a particular subscription result set SHOULD be used to retrieve subsequent results when subscription results are requested in a synchronous manor. If no more results are pending, the value of the chunkToken MUST be "0".

A chunkToken is intended as a short-term aid in obtaining contiguous results across multiple API calls and is therefore likely to remain valid for only a short time. Nodes MAY establish policies on how long a chunkToken remains valid.

5.5.6 Use of keyBag in Subscription

The subscription results returned by the subscription APIs allow for the use of a structure called a keyBag. A keyBag contains a list of entity keys, which correspond to any of the core data structures (businessEntity, businessService, bindingTemplate or tModel). The keyBag has two uses.

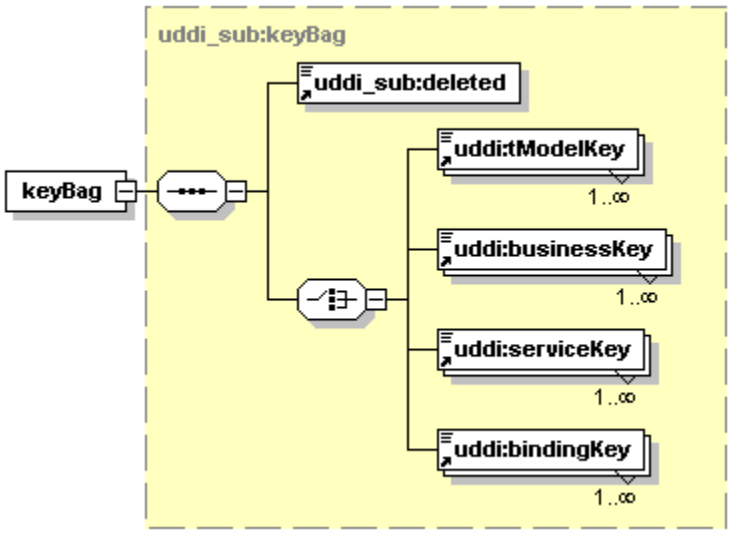
- Returning results when a "brief" format is selected, which minimizes returned information.

- Indicating entities which have been deleted, or which no longer match the subscription criteria provided with the subscription. This later situation is referred to as a "virtual delete", in that the entity in question may not actually have be deleted from the registry, but it no longer matches the criterion which the subscriber defined in the subscription for tracking registry changes. It should be noted that nodes MUST maintain information pertaining to registry changes for both forms of deletion, to be reported with subscription results for applicable subscriptions, although they MAY establish policies on how long such information is retained. Further details on the use of this structure are discussed in the relevant API sections that follow. When the keyBag is used for deleted entities, the deleted element is set to "true," and all entities listed in such a keyBag are assumed to represent deletions.

A UDDI node MUST never inform the subscriber of an entity that temporarily matched the subscription criteria but was removed or modified to no longer match the subscription criteria before the subscriber was informed of the match.

A UDDI node MAY inform a subscriber about the real or virtual deletion of an entity multiple times.

The syntax of a keyBag is shown here:



5.5.7 Subscription API functions

The APIs in this section describe how to interact with a UDDI node implementation to create and manage requests for the tracking of new and changed registry content. These APIs are synchronous and are exposed via SOAP, although the notifications they may generate are not.

The subscription APIs are:

- **delete_subscription:** Cancels one or more specified subscriptions.
- **get_subscriptionResults:** Synchronously returns registry data pertaining to a particular subscription within a specified time period.
- **get_subscriptions:** Returns a list of existing subscriptions previously saved by the subscriber.
- **save_subscription:** Establishes a new subscription or changes an existing one. Also used to renew existing subscriptions.

The OPTIONAL client API is:

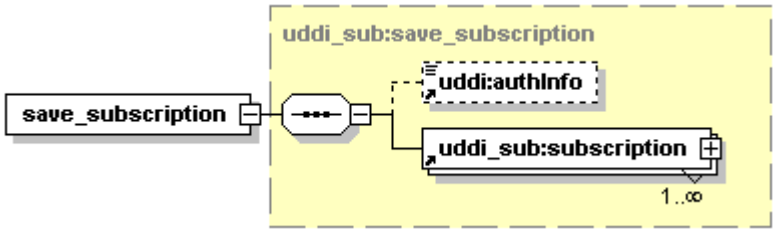
- **notify_subscriptionListener**: A node invoked API which the client implements as a subscription listener service to accept notifications containing the data that changed since notify_subscriptionListener was last invoked for a particular subscription.

5.5.8 save_subscription

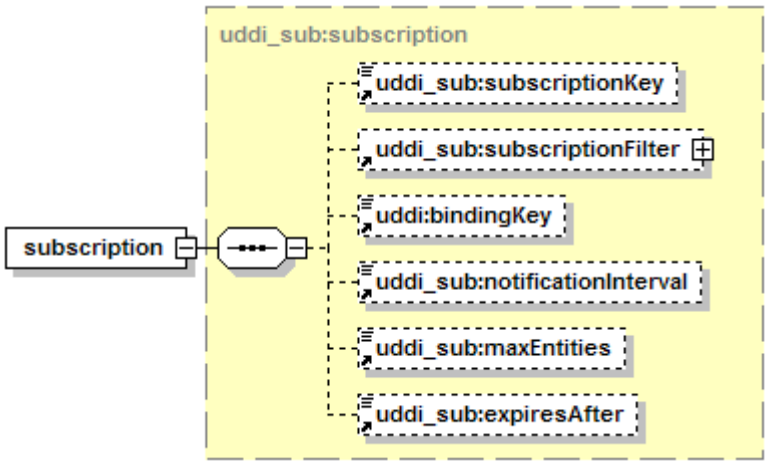
The save_subscription API registers a request to monitor specific registry content and to have the node periodically notify the subscriber when changes are available. Notifications are not returned synchronously with results for this API. Only data that matches the requested subscription criteria and which changes after the point in time that the subscription request is accepted is returned to the subscriber via a notification.

This API returns a duration for which this particular subscription is valid. Depending upon the policy of the Node, subscriptions need to be renewed before the expiration date in order to insure that they remain active. Subscriptions can also be redefined or renewed using this API. The subscriptionKey pertaining to the subscription to be renewed must be supplied in the save_subscription invocation in order to accomplish this. This allows both renewal and changes to the subscription. Invoking save_subscription automatically resets the expiration period for the subscription in question to a value based upon the node's policy.

5.5.8.1 Syntax:



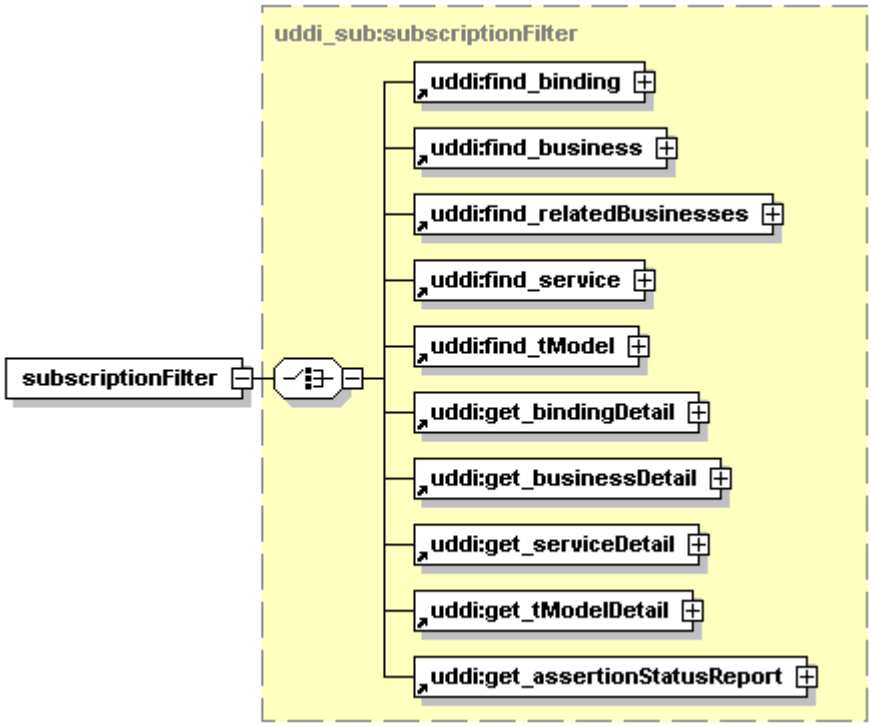
The syntax of the subscription structure is:



Attributes

Name	Use
brief	optional

The syntax of the subscriptionFilter structure is:



5.5.8.2 Arguments:

- ***authInfo***: This optional argument is an element that contains an authentication token. Registries that wish to restrict who can save a subscription typically require authInfo for this call, though this is a matter of node policy.
- ***bindingKey:*** This optional argument of type anyURI specifies the *bindingTemplate* which the node is to use to deliver notifications to subscription listeners. It is only required when asynchronous notifications are used. This *bindingTemplate* MUST define either a Web service that implements notify_subscriptionListener (see below), or an email address to receive the notifications. If a notify_subscriptionListener Web service is identified, the node invokes it to deliver notifications. If an email address is identified, the node delivers notifications via email to the address supplied. When notifications are delivered via email, the body of the email contains the body of the SOAP message, which would have been sent to the notify_subscriptionListener service if that option had been chosen. The publisher making the subscription request MUST own the bindingTemplate. If this argument is not supplied, no notifications are sent, although subscribers may still use the get_subscriptionResults API to obtain subscription results. See Section 5.5.11 *get_subscriptionResults* for details. If email delivery to the specified address fails, nodes MAY attempt re-delivery, but are not obligated to do so. Depending upon node policy, excessive delivery failures MAY result in cancellation of the corresponding subscription.
- ***brief***: This optional argument controls the level of detail returned to a subscription listener. The default is "false" when omitted. When set to "true," it indicates that the subscription results are to be returned to the subscriber in the form of a keyBag, listing all of the entities that matched the subscriptionFilter. Refer to Section 5.5.6 *Use of keyBag in Subscription,* for additional information. This option has no effect on the assertionStatusReport structure, which is returned as part of a notification when the

subscriptionFilter specifies the get_assertionStatusReport filter criteria. See the explanation of subscriptionFilter below.

- ***expiresAfter***: This optional argument allows subscribers to specify the period of time for which they would like the subscription to exist. It is of the XML Schema type xsd:dateTime. Specifying a value for this argument is no guarantee that the node will accept it without change. Information on the format of expiresAfter can be found in Section 5.5.1.1 *Specifying Durations*.
- ***maxEntities***: This optional integer specifies the maximum number of entities in a notification returned to a subscription listener. If not specified, the number of entities sent is not limited, unless by node policy.
- ***subscriptionFilter***: This argument specifies the filtering criteria which limits the scope of a subscription to a subset of registry records. It is required except when renewing an existing subscription. The get_xx and find_xx APIs are all valid choices for use as a subscriptionFilter. Only one of these can be chosen for each subscription. Notifications, based on the subscriptionFilter, are sent to the subscriber if and only if there are changes at the node, which match this criterion during a notification period. A subscriptionFilter MUST contain exactly one of the allowed inquiry elements. The authInfo argument of the specified get_xx or find_xx API call is not required here and is ignored if specified. All of the other arguments supported with each of these inquiry APIs are valid for use here.

  Specifying find_relatedBusinesses is useful for tracking when reciprocal relationships are formed or dissolved. Specifying get_assertionStatusReport can be used in tracking when reciprocal relationships (which pertain to a business owned by the subscriber) are formed, dissolved, or requested by the owners of some other business.

  For a get_assertionStatusReport based subscription, there is a specific status value, **status:both_incomplete**, defined in the XML schema. When appearing in an assertionStatusItem of a subscriptionResultsList, status:both_incomplete indicates that the publisher assertion embedded in the assertionStatusItem has been deleted from both ends.

  Note that the above handling of deleted publisher assertions is different from the case when a business entity, business service, binding template, or tModel is removed. In the latter case, the key to the entity in question is simply put inside a keyBag. A publisher assertion, on the other hand, has no key and therefore the keyBag idea is not applicable.
- ***subscriptionKey***: This optional argument of type *anyURI* identifies the subscription. To renew or change an existing subscription, a valid subscriptionKey MUST be provided. When establishing a new subscription, the subscriptionKey MAY also be either omitted or specified as an empty string in which case the node MUST assign a unique key. If subscriptionKey is specified for a new subscription, the key MUST conform to the registry's policy on publisher-assigned keys.
- ***notificationInterval***: This optional argument is only required when asynchronous notifications are used. It is of type xsd:duration and specifies how often change notifications are to be provided to a subscriber. If the notificationInterval specified is not acceptable due to node policy, then the node adjusts the value to match the next longer time period that is supported. The adjusted value is provided with the returns from this API. Also see Section 5.5.1.1 *Specifying Durations*.

5.5.8.3 Returns:

Upon successful completion this API returns a *subscriptions* structure. Included in the subscription structure(s) it MUST contain is a *subscriptionKey* (of type anyURI) that is used by

the subscriber to manage the subscription. This key is required in order to delete (unsubscribe), modify or renew the subscription. If a subscriber has multiple subscriptions, the subscriptionKey can be used to distinguish between different subscriptions. The *subscriptionKey* is also part of the data contained in the notifications returned to subscription listeners.

The subscription structure(s) returned from this API, MUST each contain an *expiresAfter* value, which has been assigned by the node. Nodes SHOULD attempt to honor the value(s) provided with the save_subscription request, but MAY modify them based on node policy. Depending upon the node's policy, the node MAY delete a subscription after it has expired.

The value of the *notificationInterval* included in the subscription structure(s) returned MAY be adjusted by the node to the value closest to that requested which is supported by its policies. Depending upon the Registry's workload a node MAY skip a notification cycle. If a cycle is skipped, the next notification sent SHOULD include information based on registry activity, which has occurred since the last notification was issued.

5.5.8.4 Caveats:

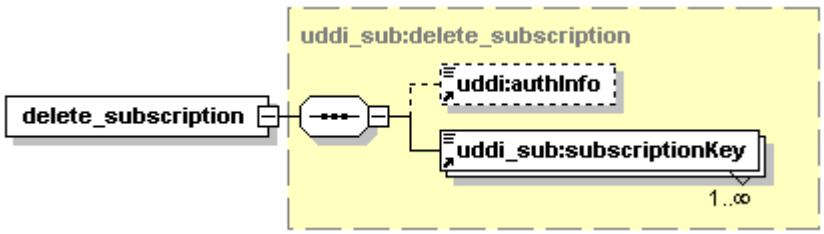
If any error occurs in processing this API call, a dispositionReport structure is returned to the caller in a SOAP Fault. In addition to the errors common to all APIs, the following error information is relevant here:

- **E_invalidKeyPassed**: signifies that an entity key value passed did not match with any known key values. The error structure signifies that the condition occurred and the error text clearly calls out the offending key.
- **E_unsupported**: signifies that one of the argument values was not supported by this implementation. The offending argument is clearly indicated in the error text.
- **E_resultSetTooLarge**: signifies that the node refuses to accept the subscription because it deems that result sets associated with the subscription are too large. The subscription criteria that triggered this error should be refined and re-issued.
- **E_accountLimitExceeded**: signifies that the request exceeded the quantity limits for subscription requests, based on node policy.
- **E_userMismatch**: signifies that an attempt has been made to use the subscription API to change a subscription that is controlled by another party. Or that the bindingTemplate specified does not belong to the publisher.
- **E_requestDenied**: signifies that the subscription cannot be renewed. The request has been denied due to either node or registry policy.

5.5.9 delete_subscription

Cancels an existing subscription.

5.5.9.1 Syntax:



5.5.9.2 Arguments:

- ***authInfo***: This optional argument is an element that contains an authentication token. Registries that wish to restrict who can delete a subscription typically require authInfo for this call, though this is a matter of node policy.
- ***subscriptionKey:*** This required argument specifies, using *anyURIs,* the subscription or subscriptions to be deleted.

5.5.9.3 Returns:

If no errors occur then an empty message is returned.

5.5.9.4 Caveats:

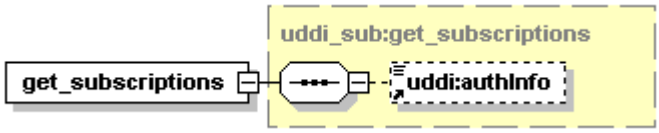
If an error occurs in processing this API call, a dispositionReport structure is returned to the caller in a SOAP Fault. In addition to the errors common to all APIs, the following error information is relevant here:

- **E_userMismatch**: signifies that an attempt has been made to use the subscription API to delete a subscription that is controlled by another party.
- **E_invalidKeyPassed**: signifies that the subscriptionKey is invalid or that the subscription has expired.

5.5.10 get_subscriptions

Returns the complete list of existing subscriptions owned by the subscriber.

5.5.10.1 Syntax:



5.5.10.2 Arguments:

- ***authInfo***: This optional argument is an element that contains an authentication token. Registries that wish to restrict who can obtain information on subscriptions typically require authInfo for this call, though this is a matter of node policy.

5.5.10.3 Returns:

This API call returns information on all of the subscriptions owned by the subscriber, together with the expiration date for each. The subscriptions structure returned contains zero or more subscription structures, each pertaining to a subscription. Only subscriptions created by the invoking subscriber are returned. See Section 5.5.8.1*, [save_subscription] Syntax*, for details on these structures.

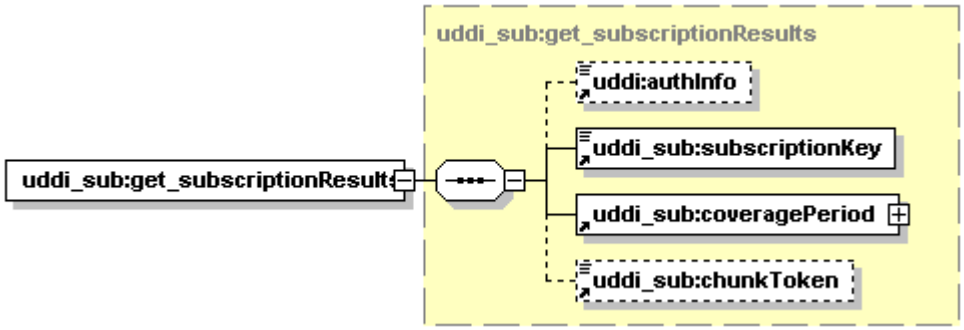
5.5.10.4 Caveats:

If any error occurs in processing this API call, a dispositionReport structure is returned to the caller in a SOAP Fault. There is no specific error information, other than the errors common to all APIs.

5.5.11 get_subscriptionResults

This API allows a subscriber to request that the information pertaining to an existing subscription be returned. This is useful, for example, to obtain historical data when a subscription is first set up or when a subscriber misses the notification normally provided by the registry. The results are returned synchronously as the response to this call. The get_subscriptionResults API can also be used as an alternative to notifications for obtaining subscription data. If this is the preference, then the subscriber SHOULD not provide a bindingKey when saving the associated subscription. See Section 5.5.8 *save_subscription*. This API is not affected by the value of the notificationInterval in the subscription.

5.5.11.1 Syntax:

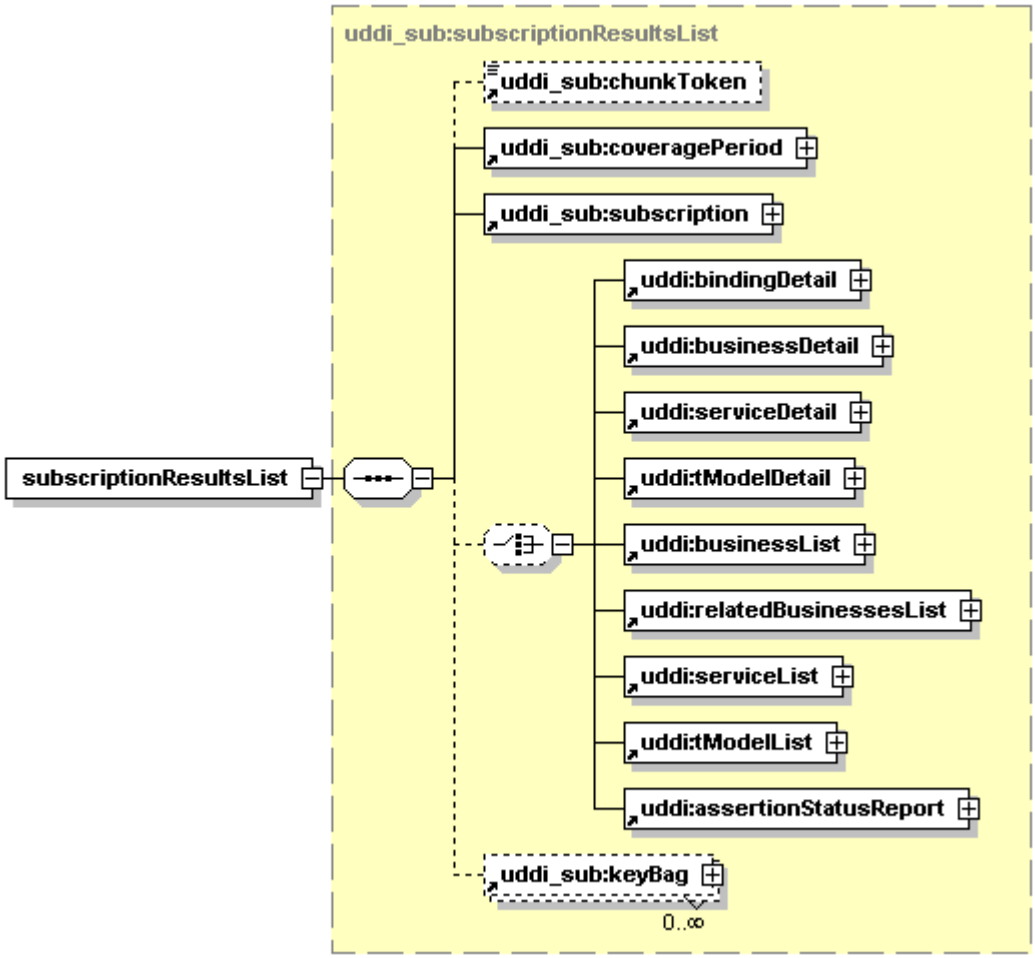


5.5.11.2 Arguments:

- ***authInfo***: This optional argument is an element that contains an authentication token. Registries that wish to restrict who can retrieve subscription data typically require authInfo for this call, though this is a matter of node policy.
- ***chunkToken***: This optional argument is used to retrieve subsequent groups of data when the first call to this API indicates more data is available. This occurs when a chunkToken is returned whose value is not "0" in the subscriptionResultsList structure described in the next section. To retrieve the next chunk of data, the value returned should be used as an argument to the next invocation of this API.
- ***coveragePeriod***: This structure defines the time period over which the most recent changes in node data are compared with the subscription criteria in order to produce the result set. It provides start and end date/time information according to the format described in Section 5.5.4 *Subscription Coverage Period*. The "current" state of registry entries pertaining to the subscription referenced by the subscriptionKey provided are returned if they were last created, changed or deleted during the specified time period.
- ***subscriptionKey:*** This required argument of type anyURI identifies the subscription for which non-recurring synchronous results are being sought.

5.5.11.3 Returns:

A subscriptionResultsList is returned whose content is determined by the coveragePeriod and the criteria in the subscription:



Attributes

Name	Use
someResultsUnavailable	optional

Subscription results MAY be chunked. See Section 5.5.5 *Chunking of Returned Subscription Data,* for more information on chunking of results. If results are chunked, then subsequent "chunks" can be retrieved using the chunkToken returned as an argument in a subsequent invocation of this API.

Note that the results returned in the subscriptionResultsList represent a snapshot of the current state of relevant entries in the registry. They are non-transactional in nature and prior states cannot be returned. Deleted entities and virtual deletes of entities, which have been changed in such a way that they no longer match the subscription criterion saved with the subscription, are returned only in the form of a keyBag structure, for which the deleted element is set to "true". A UDDI node MAY inform a subscriber about the real or virtual deletion of an entity multiple times.

The someResultsUnavailable attribute is set to "true" whenever the node has found it necessary to flush subscription results information pertaining to entity deletions (either actual or virtual) which pertain to this subscription, which have not yet been reported through prior calls to this API, or through use of the notify_subscriptionListener API described below. The period

of time which a node retains information on deletions for a given subscription is a matter of node policy.

The API used in the subscription filter determines the sort order for returned entities. By default, they will be sorted according to the behavior specified in the "returns" section of that API.

5.5.11.4 Caveats:

If an error occurs in processing this API call, a dispositionReport structure is returned to the caller in a SOAP Fault. In addition to the errors common to all APIs, the following error information is relevant here:

- **E_invalidKeyPassed**: signifies that the subscriptionKey is invalid or that the subscription has expired.
- **E_invalidValue**: signifies that the chunkToken value supplied is either invalid or has expired.
- **E_unsupported**: signifies that one of the argument values was not supported by this implementation. The offending argument is clearly indicated in the error text.
- **E_userMismatch**: signifies that, in a violation of node policy, an attempt has been made to use the subscription API to change a subscription that is controlled by another party.
- **E_invalidTime**: signifies that one or both of the values provided in the coveragePeriod range is invalid or does not define a range. The error structure signifies the condition that occurred and the error text clearly calls out the cause of the problem.

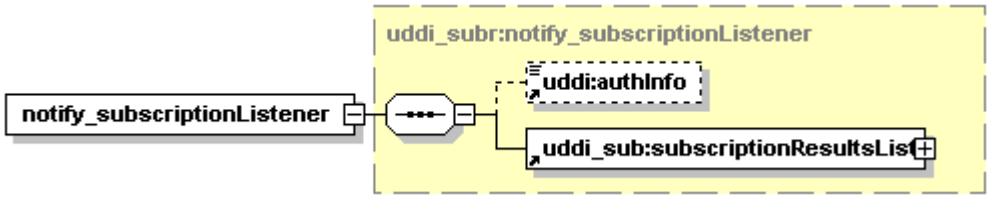
5.5.12 notify_subscriptionListener

This API, when implemented by a subscriber and specified in a subscription, enables the node to deliver notifications to subscription listeners by invoking a Web service. New, modified, and deleted data that matches the subscription is passed to notify_subscriptionListener. If the *brief* attribute of the subscription is "true", then only the relevant keys will be sent; full details of the changed data can be accomplished via the standard get_xx API's if required. If a particular item that matches the subscription criteria is deleted during the notificationInterval, or is changed in such a way that it no longer matches the criterion defined for the subscription, then these entities are included in a keyBag containing a deleted element with a value of "true".

To allow subscribers to determine whether a notification has been lost, the coverage period of the notification is included. A date/time indicating the date/time values corresponding to the start and end points of this is provided. The start date/time used in this call SHOULD align with the end date/time of the previous call and so fourth.

If the maxEntities option was specified in the save_subscription call, the response supplied via this call is limited to that number of entities. If the node cannot send all of the results in a single notify_subscriptionListener call, then the node repeatedly invokes the notify_subscriptionListener service until all information has been transmitted. In no case will the data sent to notify_subscriptionListener exceed the maximum message size per the policy of the node.

5.5.12.1 Syntax:



5.5.12.2 Arguments:

- ***authInfo***: This optional argument is an element that contains an authentication token. Subscription listener services that wish to restrict who can transmit subscription data MAY require authInfo for this call, though this is a matter of client policy.
- ***subscriptionResultsList***: This list contains the results for this notification, which consist of the result structures which are normally returned for standard find_xx or get_xx APIs, based upon the criteria saved in the subscriptionFilter for the subscription which is generating this notification. Note that the chunkToken is not returned with this structure for this API. The subscriptionResultsList also contains a coveragePeriod structure which defines the time period over which the node data is compared with the subscription criterion in order to produce the result set. It provides the start and end date/time information according to the format described in Section 5.5.4 *Subscription Coverage Period*. The "current" state of registry entries pertaining to the subscription referenced by the subscriptionKey provided are returned if they were last changed during the specified time period. See Section 5.5.11.3 *Returns* for more information on the subscriptionResultsList's content.

5.5.12.3 Returns:

Upon successful completion, notify_subscriptionListener returns an empty message. Note that this is being returned by the client supported API.

5.5.12.4 Caveats:

If an error occurs in processing this API call, a dispositionReport structure is returned to the caller in a SOAP Fault. In addition to the errors common to all APIs, the following error information is relevant here:

- **E_fatalError:** signifies the client's failure to receive notification data. The node is not obligated to retry.

5.6 Value Set API Set

Whenever a keyedReference is involved in a save operation it may be checked to see that it is valid. Similarly, a keyedReferenceGroup element that is involved in a save operation may also be checked to ensure that it is valid. Checking is performed for tModels that are deemed to be "checked", as determined by the policy of the UDDI registry.

UDDI provides the ability for third parties to register value sets, and then control the validation process used by UDDI to perform such checks. UDDI registries MAY support caching of these external value sets. UDDI registries MAY also support external validation. Node and registry policies determine the manner in which validation of references to external value sets is performed. The APIs in this section can be used by UDDI registries and nodes in their validation policies.

Third parties that want to provide an external checking capability may be required by the UDDI registry to implement a Web service in the same manner that UDDI does (e.g. using SOAP for message passing using literal encoding) that exposes a single method named validate_values. The interface for validate_values is described here.

In some cases a node may desire to eliminate or minimize the number of calls to external validation Web services. It can do so by caching valid values for those external value sets that allow caching of their values. A node has two normative options for obtaining the set of valid values. One is to periodically obtain the set of valid values from those value set providers that implement a Web service that handles the get_allValidValues API. This API is described below. The other method of obtaining a cache of valid values is to accumulate the valid values from successful calls to validate_values.

5.6.1 Value Set Programming Interfaces

The Application Programming Interfaces in this section represent capabilities that a UDDI registry MAY use to enable validation of references to value sets. Registry policy determines which external value sets are supported and how. See Section 9.4.19 *Value Set Policies* and Section 9.6.5*Value Sets* for more information on registry support of external value sets. These SOAP messages all behave synchronously.

The publicly accessible APIs that are used to support external value set validation are:

- **validate_values**: Used by nodes to allow external providers of value set validation Web services to assess whether keyedReferences or keyedReferenceGroups are valid. Returns a dispositionReport structure.
- **get_allValidValues**: Used by nodes that support caching of valid values from cacheable checked value sets to obtain the set of valid values. Returns an empty message or a dispositionReport structure.

Registry policy may require value set providers that offer one of these Web services to publish the bindingTemplate for the service and the tModel for the value set in a particular way so that the proper Web service can be discovered. See Section 9.6.5 *Value sets* for more information. When a value set provider offers one of these Web services, a tModel for the checked value set SHOULD be published in any registry the provider wishes to offer it, and a bindingTemplate SHOULD be published for the Web service(s) the value set provider offers for the checked value set. The tModel SHOULD have categorizations from the uddi-org:types category system to indicate the type of value set (*categorization*, *identifier*, *relationship*, *categorizationGroup*), that it is checked (*checked*), and, if the value set provider allows validation to occur against node caches of valid values, the *cacheable* categorization should also be provided.

In order for a value set to be considered checked, the tModel MUST first be categorized with the checked value from the uddi-org:types category system. The decision to check such value sets is a registry and node policy decision.

If a value set tModel is categorized as checked, then in response to attempts to publish a keyedReference which uses the checked tModel, nodes MUST either perform the required validation, or return E_unsupported.

The tModel should also have a categorization reference to the bindingTemplate of the get_allValidValues or validate_values Web service that the value set provider designates, using the uddi-org:validatedBy category system. See Section 11.1.1 *UDDI Types Category System* and Section 11.1.7 *Validated By Category System* for more information.

The bindingTemplate for the get_allValidValues or the validate_values Web service SHOULD reference in its tModelInstanceDetails the appropriate value set API tModel (Section 11.2.7 *Value Set Caching API tModel* or Section 11.2.8 *Value Set Validation API tModel*) as well tModels for all of the value sets the service applies to.

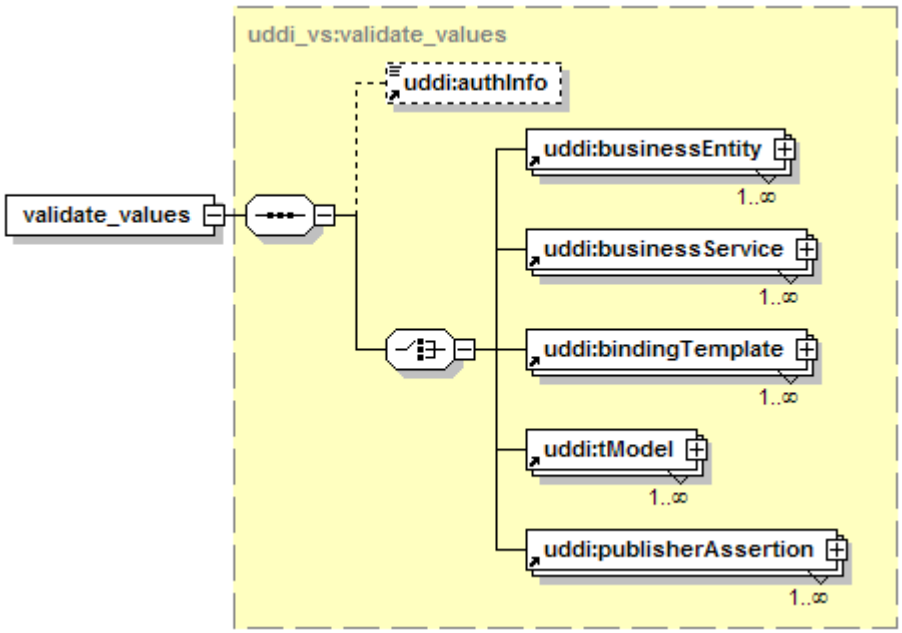
5.6.2 validate_values

A UDDI node that supports external validation sends the validate_values API to the appropriate external Web service, of which there is exactly one, whenever a publisher saves data that uses a keyedReference or keyedReferenceGroup whose use is regulated by the external party who controls that Web service. For purposes of discussion, the identifier, category, and relationship type systems that the keyedReference elements refer to are called checked value sets. The category group systems that the keyedReferenceGroup elements refer to are similarly called checked category group systems.

The normal use for checked value sets is to verify that specific values (checking the keyValue attribute of values supplied) exist within the value set. For certain value sets the value set provider may further restrict the use of a value based on a contextual evaluation of the passed data. The provider may do enable this contextual checking by offering a validation Web service.

Validation algorithms for checked category group systems similarly verify that the contents of the keyedReferenceGroup elements form a valid set according to the validation algorithm for the checked category group system. Frequently such validation ensures that the value sets identified in contained keyedReferences are allowed to participate in the category group system.

5.6.2.1 Syntax:



5.6.2.2 Arguments:

The UDDI node that is calling validate_values MUST pass one or more businessEntity elements, one or more businessService elements, one or more bindingTemplate elements, one or more tModel elements, or one or more publisherAssertion elements as the sole argument to this Web service. The one or more elements passed represents the outermost UDDI data structure(s) being passed within a save_business, save_service, save_binding, save_tModel, add_publisherAssertion, or set_publisherAssertions API call. Multiple elements of the same type may be passed together if multiples are included in the same save invocation.

The optional authInfo argument is an element that contains an authentication token. An authentication token is obtained using the get_authToken API call or through some other means external to this specification. Providers of validate_values Web services that serve multiple registries and providers that restrict who can use their service may require authInfo for this API.

5.6.2.3 Behavior

The called Web service for a checked value set performs validation on all of the keyedReferences or keyedReferenceGroups that are associated with the value sets the Web service is authorized to check. This can involve merely checking that the *keyValue* values supplied are good for the given value set (as signified by the embedded keyedReference tModelKey values). Other types of validation as desired may be performed, including context sensitive checks that utilize the information passed in the entity being saved.

The entity being saved may contain multiple references to values from the value set(s) that the validation Web service is authorized to validate. When the entity being saved is a businessEntity, contained businessService and bindingTemplate entities may themselves reference values from the authorized value sets as well. All references to values that are associated with the value set(s) that the validation Web service is authorized to check MUST be validated without regard to their placement in the entity being saved.

If the external value set and the node both support caching of valid values, the node may not invoke validate_values if it already knows that the referenced values are valid, through checking its cache.

A checked category group system is treated in the same manner as a checked value set. The tModelKey associated with the keyedReferenceGroup identifies the checked category group system. A node may be able to validate a reference to a cacheable checked category group system without calling validate_values if it can determine using its cache that the tModelKey attributes from the keyedReference elements contained in the keyedReferenceGroup are allowed for the category group system.

5.6.2.4 Returns:

If all values referenced in the entity being saved are valid from the value set(s) or category group system(s) that the validation Web service is authorized to validate, the proper response is an empty message.

5.6.2.5 Caveats:

If any error is found, or the called Web service needs to signal that the information being saved is not valid based on the validation algorithm chosen by the external Web service provider, then the Web service MUST raise a SOAP Fault as specified in Section 4.8 *Success and Error Reporting.*

When an error is signaled in this fashion, the UDDI node MUST reject the pending change and return to the original caller the same SOAP fault data returned by the validation Web service. The error codes indicate one of the following reasons, and the error text clearly indicates the keyedReference or keyedReferenceGroup data that is being rejected and the reason it is being rejected.

- **E_invalidValue**: One or more of the keyValues in the keyedReference or keyedReferences in the keyedReferenceGroup supplied failed validation. Only the first error encountered need be reported.
- **E_valueNotAllowed**: The values may be valid, but are not allowed contextually.

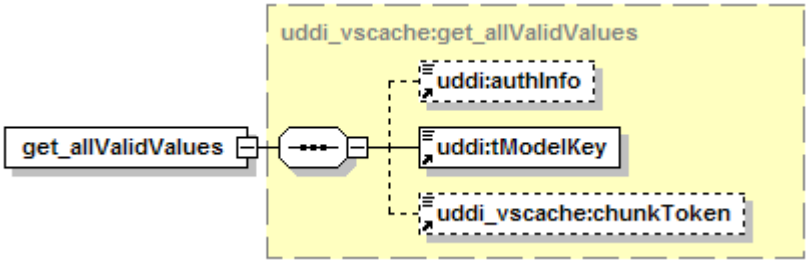
5.6.3 get_allValidValues

A UDDI node that supports external value sets MAY invoke a get_allValidValues Web service offered by a value set provider that has granted permission to that registry to cache the valid values for that value set. The external value set provider MAY offer the get_allValidValues Web service and the UDDI node MAY use it. The normal use is to return a full set of valid values for the identified value set. If the value set provider determines there are too many values to return in one chunk, the set of valid values may be returned in chunks.

Registry policy may require the value set provider that offers a get_allValidValues Web service to republish its value set tModel when the cache should be re-acquired by participating nodes. See Section 9.6.5 *Value Sets* for more information.

get_allValidValues can similarly be used to obtain the set of tModelKeys for value sets that can participate in a cached category group system.

5.6.3.1 Syntax:

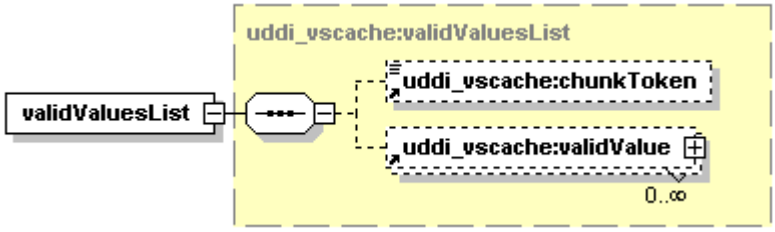


5.6.3.2 Arguments:

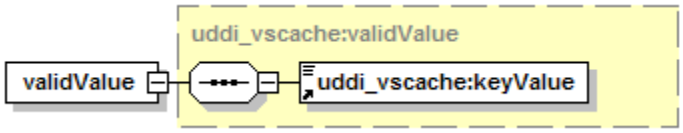
- ***tModelKey***: A required *uddiKey* value that identifies the specific instance of the tModel which describes the value set or category group system for which a Web service to get all valid values has been provided. It uniquely identifies the category, identifier, or category group system for which valid values are being requested.
- ***chunkToken***: Optional element used to retrieve subsequent groups of data when the first invocation of this API indicates more data is available. This occurs when a chunkToken is returned whose value is not "0" in the validValuesList structure described in the next section. To retrieve the next chunk of data, the chunkToken returned should be used as an argument to the next invocation of this API.
- ***authInfo***: An optional element that contains an authentication token. Authentication tokens are obtained using the get_authToken API call or through some other means external to this specification. Providers of get_allValidValues Web services that serve multiple registries and providers that restrict who can use their service may require authInfo for this API.

5.6.3.3 Returns

A validValuesList structure is returned containing the set of valid values for the external category or identifier system. The list MUST contain a chunkToken if the Web service provider wishes to provide the data in packets. The validValuesList has the form:



And its contained validValue element has the form:



5.6.3.4 Behavior

The called Web service returns the set of valid values in a *validValuesList* on success. This structure lists every valid value associated with the value set or category group system that is described by the tModelKey provided. In the event too many values exist to be returned in a single response (i.e., the message size exceeds the maximum number of bytes allowed by the UDDI registry), or the value set provider wants to supply the values in multiple packets, then the *validValueList* includes the *chunkToken* element and the API can be re-issued to get the remaining valid values.

5.6.3.4.1 Chunking of valid values

If the value set provider determines that there are too many values to be returned in a single group, then the provider SHOULD provide a chunkToken with the results. The chunkToken is a string based token which is used by the value set provider to maintain the state of the set of values for a particular caller, when these results are chunked across multiple responses. Providers should establish their own policies for determining the content and format of the chunkToken. The chunkToken returned with a particular value set result set SHOULD be used to retrieve subsequent results. If no more results are pending, the value of the chunkToken will be "0" or the chunkToken will be absent.

A chunkToken is intended as a short-term aid in obtaining contiguous results across multiple API calls and is therefore likely to remain valid for only a short time. Value set providers may establish policies on how long a chunkToken remains valid.

5.6.3.5 Caveats:

If any error occurs in processing this API, a dispositionReport structure MUST be returned to the caller in a SOAP Fault. See Section 4.8 *Success and Error Reporting.* The following error information is relevant:

- **E_invalidKeyPassed**: Signifies that the tModelKey passed did not match with the uddiKey of any known tModels. The details on the invalid key SHOULD be included in the dispositionReport element.
- **E_noValuesAvailable**: Signifies that no values could be returned.
- **E_unsupported**: Signifies that the Web service does not support this API.

- **E_invalidValue**: Signifies that the chunkToken value supplied is either invalid or has expired.

6 Node Operation

This chapter defines the normative behavior required of an operator hosting a UDDI node. It outlines the operational parameters and requirements that a node MUST follow. It also provides guidance when first bringing a node online. The intended audience for this chapter is someone intending to implement and host a UDDI node.

Note that this chapter alone is not sufficient to understand how to implement and host a node. Refer to each of the preceding chapters, in order to understand the full scope of the UDDI specification. They provide the normative behavior in terms of how the APIs work. Considerations regarding intra-registry operation (such as replication between nodes) and inter-registry operation (such as publishing data across multiple registries) are dealt with in Chapter 7 *Inter-Node Operation* and Chapter 8 *Publishing Across Multiple Registries* respectively. Also, node implementers must be cognizant of the policy decisions that they must make; the list of policy-related decisions can be found in Chapter 9 *Policy*.

This chapter addresses only the operational specifics that must be followed when hosting a node, regardless of whether that node exists as the sole node in a registry or takes part in a multi-node environment.

6.1 Managing Node Contents

This section provides procedures and requirements for managing and maintaining the information within the UDDI registry.

6.1.1 XML Requirements

Given the use of XML, XML Schema, SOAP, and XML-Dsig within UDDI, a node must take care to adhere to these standards when processing data. Specific additional requirements layered upon these standards are included in the UDDI specification. Each node is responsible for implementing these additional requirements as well as those defined by the published standards.

6.1.1.1 Processing by XML Schema Assessment

UDDI nodes MUST assess the validity of the XML elements that comprise the API requests they receive. This SHOULD be carried out by means of appealing to the 2nd or 3rd approaches enumerated in Section 5.2 *Assessing Schema-Validity* of XML Schema Structures (http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/#validation_outcome). API request elements which, having been so assessed are not found to have a [validity] property (http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/#sic-e-outcome) of *valid,* MUST not be further processed by the node. An indication of this fact MUST be reported to the client by returning an error.

Note that a side effect of processing by XML Schema Assessment is that whitespace in elements and attributes is normalized (http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/section-White-Space-Normalization-during-Validation) in a well-defined manner.

6.1.1.2 Normalization and Canonicalization

UDDI registries provide publishers with the ability to digitally sign entities they publish, and inquirers with the ability to validate the digital signatures on published material. In order for this to be possible, publishers and registries MUST handle "normalization" and "canonicalization" as described in Section 4.6.1.1 *Normalization and Canonicalization*.

6.1.2 Key Generation and Maintenance

At the core of the UDDI specification is the ability to uniquely identify entities in a UDDI registry.

6.1.2.1 Key Uniqueness

Each key stored by a node must be verified to be unique. The node must not allow the generation or storage of a key that is already present somewhere in the registry. This applies to both the node where the entry was originally saved, and, in the event the node is a part of a multi-node registry, any other node in the registry. See Section 4.4 *About uddiKeys* for an extensive discussion around considerations in key generation and Chapter 9 *Policy* for more on policy decisions a node must make around key generation.

6.1.2.2 Node Generated Keys

When a node generates a uuidKey for an entity, it must make certain that the process of creating these is a correct one. Nodes SHOULD use the time-based or random number based UUID generation algorithm as defined at http://uddi.org/pubs/draft-leach-uuids-guids-01.txt.

6.1.3 Updates and Deletions

When an entity that contains other entities is updated, the entire content of the updated entity, including contained entities, is replaced. When making updates to the registered information, the integrity of the overall registry must be maintained.

In particular, when deleting information from the UDDI registry, the following atomicity must be maintained:

- When the deletion of an entity occurs and that entity contains other entities (i.e., a businessService within a businessEntity), all contained entities MUST also be deleted.
- Any referenced entities (i.e., a tModel reference) MUST NOT be deleted.
- tModel deletion (more accurately described as deprecation or "hiding") behavior is different from that of the other UDDI entities. The result of a tModel deletion request is that it is stored in a deprecated state.

6.2 Considerations When Instantiating a Node

When first bringing a node online, several steps should be followed.

6.2.1 Canonical tModel Bootstrapping

A node MUST provide the canonical UDDI tModels (outlined in Chapter 11 *Utility tModels and Conventions*) for entities to reference. The node MAY acquire these canonical tModels either through performing an import of these tModels, through participating in a replication topology or through other means.

6.2.2 Self-Registration of Node Business Entity

6.2.2.1 Normative Modeling of Node Business Entity

A node MUST register itself in the UDDI registry of which it is a part, so that it can be uniquely distinguished within a registry. This UDDI entry is known as the "Node Business Entity". The node MUST categorize its Node Business Entity with the uddi:uddi.org:categorization:nodes tModel using the keyValue, "node":

```
<categoryBag>
    <keyedReference
        tModelKey="uddi:uddi.org:categorization:nodes"
        keyName=""
```

```
            keyValue="node"/>
    </categoryBag>
```

This checked category system is only permitted to be used by the node itself. If another publisher attempts to save a categoryBag that has a reference to the "uddi:uddi.org:categorization:nodes" category system, the node MUST return an error. This checking guarantees the ability to query a single node and determine all the nodes that participate in a given registry.

6.2.2.2 Recommended Modeling of Node Business Entity

A node is composed of Web services that implement one or more UDDI API sets. A node SHOULD model those Web services within its Node Business Entity. The modeling of the Node Business Entity SHOULD include the following characteristics:

- Each bindingTemplate that describes an implementation of a UDDI API set SHOULD use tModelInstanceInfos that reference the API tModels of the API sets the implementation supports (e.g., uddi-org:inquiry_v3). See Chapter 9 *Policy* and Section 11.1.9 *UDDI Registry API tModels*.
- Each bindingTemplate provided SHOULD model the transport protocols and security protocols supported by the implementation it describes.
- Each bindingTemplate provided SHOULD model the policies of the given node. One option is to provide those policies in a document that can be found via the overviewURL of the tModelInstanceInfo of a given bindingTemplate. Another option is to model those policies using instanceParms, categorization schemes or other UDDI constructs.

Because the UDDI data model offers great flexibility in how the modeling of services is achieved and because the context in which a node might exist varies greatly, normative mandates on the modeling of a Node Business Entity are inappropriate. However, it may be the policy of a given registry that each node participating in that registry must model its Node Business Entity according to a given template or pattern.

6.3 User Credential Requirements

6.3.1 Establishing User Credentials

A UDDI node MAY require users to establish an account with the node, before they are allowed to utilize some or all of the services at the node. The process for registration is determined by policy and may be node-specific.

For registries or nodes that enforce a policy relating particular publishers as owners of particular datum, it is essential that there is a mechanism to identify the publisher of each entity published at the node. Although this data is not reflected in the schema provided by UDDI, it should be stored and obtainable by a node for entities for which it has custody of. See Section 1.5.8 *Data Custody* and Section 1.5.6 *Person, Publisher and Owner*.

6.3.2 Changing Entity Ownership

A UDDI node SHOULD provide an interface to permit a user to transfer ownership of data it currently owns to another user within that node. An API for this procedure is given in Section 5.4 *Custody and Ownership Transfer API*.

6.4 Checked Value Set Validation

This section describes the normative node behavior with respect to value set references. Much of the node behavior in this area is not normative, but instead is driven by registry and node policy.

For more on external validation checking, see Section 5.6 *Value Set API* and Chapter 9 *Policy*.

6.4.1 Normative behavior during saves

Every time a publisher saves an entity that has a keyedReference associated with a checked value set, the node MUST perform validation in a manner that is acceptable to the value set provider, or MUST reject the reference with an E_unsupported error code. How a UDDI node determines that the referenced value set is checked, where that value set is hosted, locates and invokes the validation algorithm, determines that a validation algorithm is acceptable, and deals with unavailability of the validation algorithm is all driven by registry, and in some cases, node policy.

When a UDDI node encounters a reference to a checked value set that it validates it MUST:

- Perform validation for each top-level entity being saved.
- Inspect tModel references in categoryBag, identifierBag or publisherAssertion keyedReferences, and keyedReferenceGroups to determine if references are to be checked.
- Fail the save when any keyedReference fails validation.
- Pass along error information for E_ invalidValue and E_valueNotAllowed errors from the validation algorithm to the caller of the save operation.
- Return E_unvalidatable if the validation algorithm is unavailable.

A UDDI node that encounters a keyedReference for a checked value set that it validates MUST do so using one of the following methods:

- Invoke the validation algorithm once for each individual reference, or once for the for the collective set of references to all value sets that share the same validation algorithm
- Perform validation itself on cached values for checked value sets that the value set provider has allowed to be cached.
- Grandfather previously validated value references when the validation algorithm is not available such that existing keyedReferences are considered still valid and are not re-checked while the validation algorithm is unavailable. Nodes MAY allow previously validated keyedReferences associated with that unavailable value set to be (re)published without failing the save (e.g. the references are grandfathered and remain valid even though they can't be checked). This gives nodes the opportunity to not penalize those publishers that attempt to reference a value set marked unvalidatable. However, new value references an unavailable value sets MUST be rejected.

6.5 HTTP GET Services for UDDI Data Structures

A node may offer an HTTP GET service for access to the XML representations of UDDI data structures. If a node offers this service, the URLs should be in a format that is predictable and uses the entity key as a URL parameter.

The RECOMMENDED syntax for the URLs for such a service is as follows:

If a UDDI node's base URI is http://uddi.example.org/mybase, then the URI http://uddi.example.org/mybase?<entity>Key=uddiKey would retrieve the XML for the data structure whose type is <entity> and whose key is uddiKey. For example, the XML representation of a tModel whose key is "uddi:tempuri.com:fish:interface" can be retrieved by using the URL http://uddi.example.org/mybase?tModelKey=uddi:tempuri.com:fish:interface.

In the case of businessEntities, the node MAY add these URIs to the businessEntity's discoveryURLs structure, though this is NOT RECOMMENDED behavior as it complicates the use of digital signatures.

7 Inter-Node Operation

This chapter defines the normative behavior of UDDI nodes interacting as a single UDDI registry. It outlines the operational parameters and requirements that UDDI nodes and UDDI registries must follow. These parameters and requirements are policies that a registry must establish and that the nodes must enforce. This chapter also describes the OPTIONAL replication protocol for propagation of UDDI data among nodes of a registry. The intended audiences for this chapter are registry administrators and node operators intending to implement and host a multi-node UDDI registry.

Considerations regarding inter-registry operation, such as data import and export between affiliated registries, are detailed in Section 8.2 *Data Management Policies and Procedures Across Registries*.

7.1 Inter-Node Policy Assertions

Any set of nodes composing a single UDDI registry MUST enforce a set of policies for keying, data management and value sets as defined in Chapter 9.

This chapter focuses on the replication protocol that is appropriate to a multi-node registry that uses a single-master data model and assumes the data custody policy in the following section.

7.1.1 Data Custody

Registries that use the replication protocol defined in Section 7.4 Replication API Set MUST enforce the data custody policy specified in Section 1.5.6 *Data Custody*.

Portions of the Custody Transfer process discussed in Section 5.4 *Custody and Ownership Transfer API* are designed to utilize UDDI Replication.

7.2 Concepts and Definitions

Where two or more nodes are integrated into a registry, the use of the replication API described in this chapter allows the registry to be viewed as a single logical entity. A registry designed in this way supports uniform access to a complete set of registry data from any node within the registry. The goal of replication is to facilitate the establishment and maintenance of a single consistent shared set of registry data. Replication latency notwithstanding, all nodes in a registry should at all times contain common content.

This chapter describes the data replication process and programmatic interface required to achieve complete data replication in UDDI registries composed of more than one node. The replication process makes use of Extensible Markup Language (XML) and Simple Object Access Protocol (SOAP) specification for using XML in simple message-based exchanges as described in Section 7.4 *Replication API Set*.

In general terms this section describes the replication API's and behavior required of any node that intends to support UDDI replication. This function may be thought of as satisfying the following base requirements:

- Support the addition of a node, by supplying to it an image of the current registry contents.
- Support periodic replication between the nodes that compose a registry.
- Support recovery from errors encountered during replication processing.

Note: Please refer to Chapter 2 *UDDI Schemas* for the reference to the UDDI XML Schema and to the UDDI Replication XML Schema files.

7.2.1 Update Sequence Number

Each node SHALL maintain a strictly increasing register known as its *Originating Update Sequence Number* (USN). An originating USN is assigned to a change record at its creation by a node. The originating USN SHALL NEVER decrease in value, even in the face of system crashes and restarts. UDDI nodes MUST NOT rely on an originating USN sequence increasing monotonically by a value of "1". Gaps in a node's originating USN sequence MUST be allowed for as they are likely to occur in the face of system crashes and restarts.

While processing changes to the Registry as a result of performing UDDI Replication, all replicated data MUST be assigned an additional unique and locally generated USN register value – a *local USN*.

The originating and local USN registers MUST be sufficiently large such that register rollover is not a concern. For this purpose, UDDI nodes MUST implement a USN of exactly 63 bits in size.

Note that it is semantically meaningless to compare USNs that have been generated on different nodes; only USNs generated on the same node may be meaningfully compared to each other.

NO change record MAY have a USN equal to 0 (zero).

7.2.2 Change Records

When a publisher changes a specific datum at a node, the node will create a *change record* that describes the details of the change.

Each change record contains the following information:

- The unique key of the *originating node* at which the change record was initially created.
- An *originating USN* with which the change record is assigned at its creation by its originating node.
- A payload of data that conveys the semantics of the change in question. These are elaborated and specified later in this chapter.

The UDDI replication process defines how change records are transmitted between nodes of a registry. Each step of replication involves the transmission of such records from one node to another, say from Node A to Node B. As Node B receives change records from Node A, Node B assigns each incoming record with a fourth piece of (Node-B-generated) information, a *local USN*.

It is RECOMMENDED that within an implementation, a node should first process a new or updated record and then increment its originating and local USN registers. This process assures that the USN values remain unique.

Should an implementer choose to record change records generated by an implementation within a *Change Record Journal*, the local USN and the originating USN values it assigns to a given change record MAY be set to an identical USN value. Thus, all change records ever processed by a node can be sequenced in order of the time of arrival via the replication stream by sorting on the local USNs values of change records it holds.

A node that is ready to initiate replication of change records held at another node within the registry uses the get_changeRecords message. Part of the message is a *high water mark vector* that contains for each node of the registry the originating USN of the most recent change record that has been successfully processed by the invocating node. The effect of receiving a get_changeRecords message causes a node to return to the calling node change records it has generated locally and processed from other nodes constrained by the directives of the high water mark vector specified. As such, by invoking get_changeRecords a node obtains from its *adjacent* node all change records (constrained by the high water mark vector) the adjacent node has generated locally or successfully processed from other nodes participating in the replication topology. What constitutes an adjacent node is governed by the replication *communication graph*. Replication topology is controlled via a Replication Configuration Structure. Amongst other parameters, the Replication Configuration Structure identifies one unique URL to represent the replication point, soapReplicationURL, of each of the nodes of the registry.

Upon receiving a get_changeRecords message, a node MUST return change records strictly in increasing order of its local USN values. This property, together with the rules by which local USNs are assigned, provides the following guarantee: Suppose a publisher, whose data is held at node B (that is, Node B is the custodian of that data) changes its data. Node B originates a change record *cr*. Then it is guaranteed that any other node that receives change record *cr* will have previously received all changes which on Node B had a local USN less than the local (and originating) USN of *cr.* This, however, is true *only* of changes *cr* that Node B originates.

This important cause and effect relationship is relied upon in several places in the UDDI Replication design, notably the algorithm by which information is ultimately deleted from the UDDI registry.

7.2.3 Change Record Journal

Accurate replication of data within a UDDI registry relies on accurate and faithful creation, transmission, and retransmission of change records between all nodes of the UDDI registry. A change record originated by a node is the authoritative reference for change information propagation. It is critical that change records are not inadvertently altered as they are conveyed from the originating node through intermediate nodes to reach other nodes in the UDDI registry.

To that end, each node SHOULD create and maintain as part of their internal implementation a *change record journal* that explicitly records verbatim the XML text of change records as they are received from other nodes. This journaling may be performed before or after standard XML parsing of the change record.

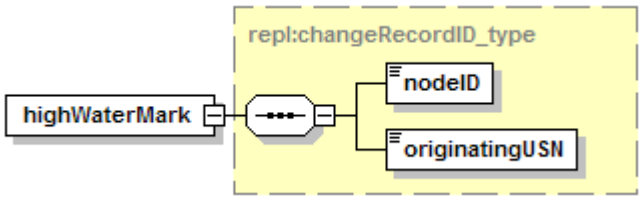
Implementers may find it convenient to also place their own change records in their change record journal as described above.

7.2.4 High Water Mark Vector

Each node maintains state information in the form of a *high water mark vector* that contains the originating USN of the most recent change record that has been successfully processed by each node of the registry. This vector has one entry for each node that can now or has ever introduced change records into the replication stream. Each entry contains the following information:

- ***nodeID***, that provides the unique key of a node in the replication communication graph, and
- ***originatingUSN***, that provides the originating USN of the most recent change associated with the node identified by operatorNodeID that has been successfully consumed. Since changes originating from a given node are always originated and thus consumed in order, this will necessarily normally be the largest originating USN that the calling node has successfully consumed from the node identified.

A consuming node MAY reset the originating USN to another value that had been previously been requested for a given node. This may occur due to the need to obtain change records from other nodes as part of a recovery operation following a system failure.



7.2.5 Replication Messages

Replication processing consists of the notification of changes that are available and then the subsequent "pulling" of changes from one of the nodes within the registry. A node within the registry MAY advertise that changes are available at that node. The advertisement of changes available includes sufficient information so that another node within the registry can determine if and when it should pull the changes from the offering node to itself for replication processing.

UDDI Replication defines four messages. The first two presented here are used to perform replication and issue notifications. These are:

- ***get_changeRecords*** - This message is used to initiate the replication of change records from one node to another. The invoking node, wishing to receive new change

records, provides as part of the message a high water mark vector. This is used by the replication source node to determine what change records satisfy the caller's request.

- ***notify_changeRecordsAvailable*** - Nodes can inform others that they have new change records available for consumption by replication by using this message. The notify_changeRecordsAvailable message is the predecessor to the get_changeRecords message.

Figure 3 depicts the use of the get_changeRecords and notify_changeRecordsAvailable messages to carry out the process of replicating changes between UDDI nodes.

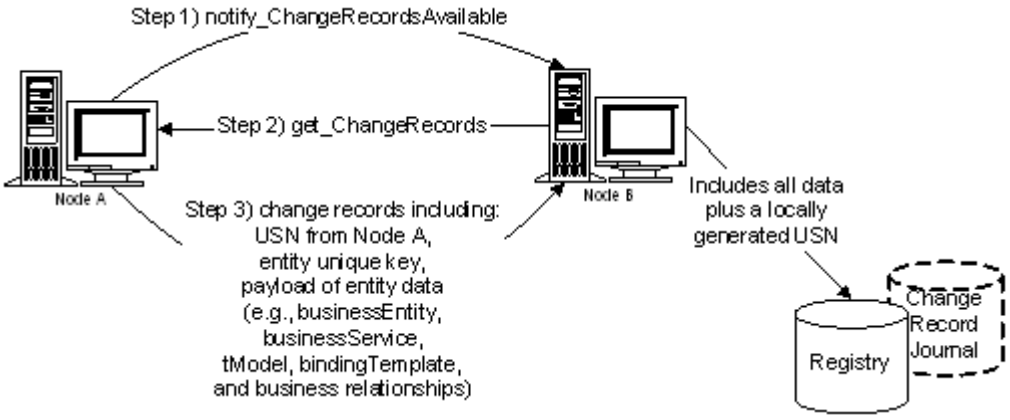


Figure 3 - Replication Processing

Two ancillary messages are also defined to support UDDI Replication. These are:

- ***do_ping*** - This UDDI API message provides the means by which the current existence and replication readiness of a node may be obtained.
- ***get_highWaterMarks*** - This UDDI API message provides a means to obtain a list of highWaterMark elements containing the highest known USN for all nodes in the replication communication graph.

7.2.6 Replication Processing

Replication SHOULD be configured so that in the absence of failures, changes propagate throughout the UDDI registry within a set amount of time. This requirement means that get_changeRecords requests MAY have to be sent in some scheduled manner.

For example, assume that the communications graph is a cycle of 4 nodes (A, B, C, and D) such that D places get_changeRecords request to C (D>C), C>B, B>A, and then finally A>D.

In this example, A starts the Replication process. Periodically, A generates a timer event and notifies B of its high water vector; if necessary, B issues a get_changeRecords request to A, and then sends C its high water vector. This continues around the cycle (B>A, C>B, D>C, A>D), but doesn't stop there. B has not received any changes from C or D for the current period, and C has not received any changes from D.

So A continues this algorithm around the cycle again (B>A, C>B, D>C). At this point, all changes that existed when A handled its timer event have been circulated to all nodes. (Subsequent changes may have also been propagated.)

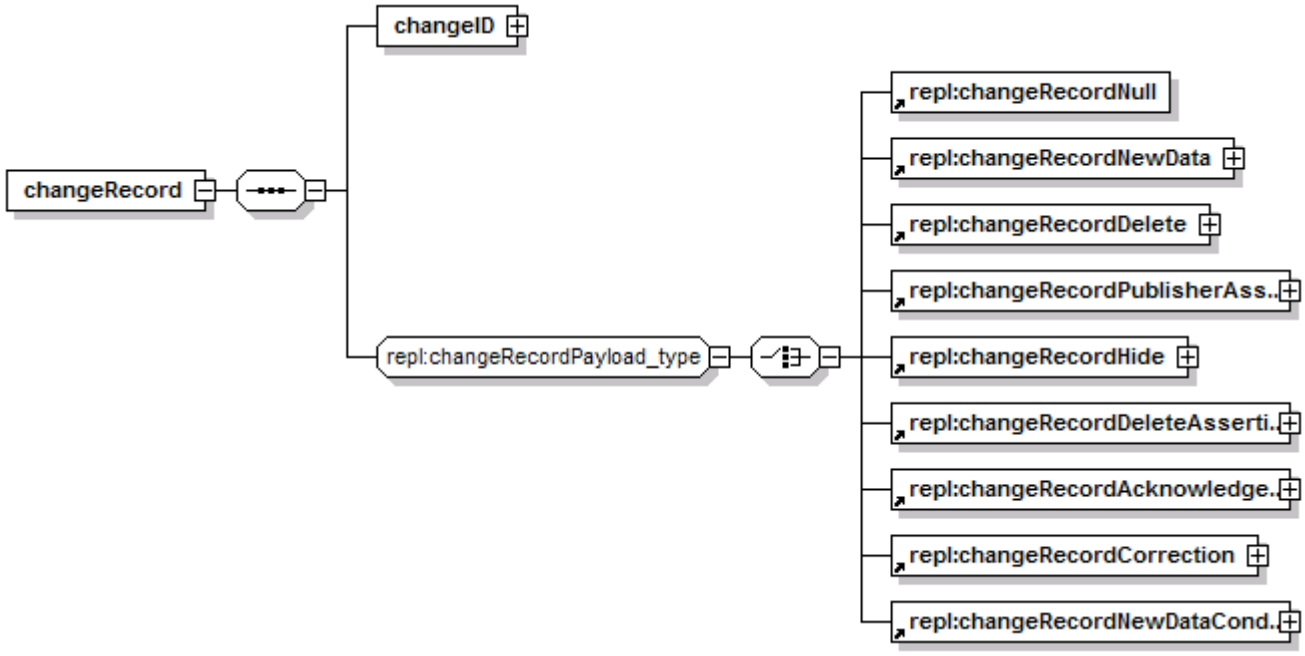
Nodes within the registry MAY do local optimizations of changes prior to replicating the changes throughout the UDDI registry. Any local optimizations performed must be invisible to other nodes within the registry.

This UDDI API provides a means to obtain a list of highWaterMark elements containing the highest known USN for all nodes in the replication communication graph.

Validation of replicated data is discussed in Section 7.7 *Validation of Replicated Data*. Error detection and processing is discussed in Section 7.6 *Error Detection and Processing*.

7.3 Change Record Structures

This section provides the definition of the various changeRecord elements defined for use for UDDI Replication. The overall changeRecord element is defined as follows.



Each change record contains a changeID that identifies the node on which the change originated and the originating USN of the change within that node. It then contains one of several allowed forms of change indication; these are elaborated below. With the exception of a changeRecordAcknowledgement type record, a changeRecord may contain an acknowledgementRequested attribute.

If present with the acknowledgementRequested value set to "true," then when each node receives a change record and successfully processes it into internal data store, that node MUST in turn originate a new changeRecord with a changeRecordPayload_type of changeRecordAcknowledgement. This is done to acknowledge the message processing success and allow that knowledge to be disseminated through the rest of the UDDI registry.

As each changeRecord element first arrives at a node, it must be assigned a local USN value from the receiving node's USN register. This local USN allows the node to maintain over time the relative order of changes it has received from others and changes it has originated locally. As was mentioned previously, when changes are replicated to others in response to a get_changeRecords request, the change records are provided in ascending order according to this local USN. However, the local USN itself never actually appears in any node-to-node data transmission.

In the event that any changeRecordPayload_type listed below is deprecated in a future version of this specification, transmissions of the change records of the deprecated changeRecordPayload_type MUST be treated as replication errors. The corresponding

handling of those replication transmission errors is specified within Section 7.6 *Error Detection and Processing*.

7.3.1 changeRecordNull

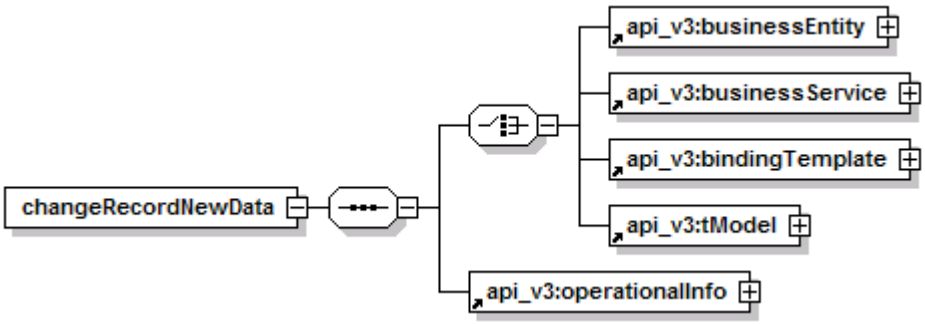
The changeRecordNull element is defined as follows:

Change records of this form do not in fact indicate any sort of semantic change. Rather, their utility largely lies in providing a convenient and safe means to exercise and test certain aspects of the UDDI replication infrastructure. In addition, a changeRecordNull to which an acknowledgement request is attached expands this testing capability of the UDDI registry.

A changeRecordNull is considered "successfully processed" once a node has received it and durably stored it in its Change Record Journal.

7.3.2 changeRecordNewData

The changeRecordNewData element is defined as follows:



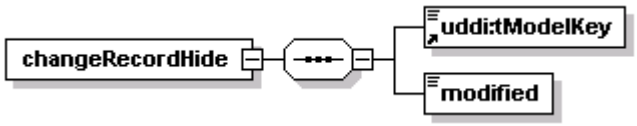
A changeRecordNewData MUST not be empty; it must contain a valid semantic piece of new data. Change records of this type provide new or updated business or modeling information that is to be incorporated. Partial updates to a datum are not provided for; rather, the entire new contents of the datum and its operationalInfo are to be provided, and these replace any existing definition of the datum and its operationalInfo with the recipient of the change record. The hidden state (i.e. the deleted attribute) must be persisted through replication to allow for a custody transfer of hidden tModels between nodes via the replication protocol.

The operationalInfo element MUST contain the operational information associated with the indicated new data. No validation other than schema assessment and presence requirements are performed by the consuming node. Specifically, the creation date may change; the creation date need not be earlier than the modification date; the modification date need not be earlier than the modified including children date.

A changeRecordNewData is considered "successfully processed" once a node has received it, assigned a local USN to it, validated it, durably stored it in its change record journal, and then successfully incorporated it into the node's data store.

7.3.3 changeRecordHide

The changeRecordHide element is defined as follows:

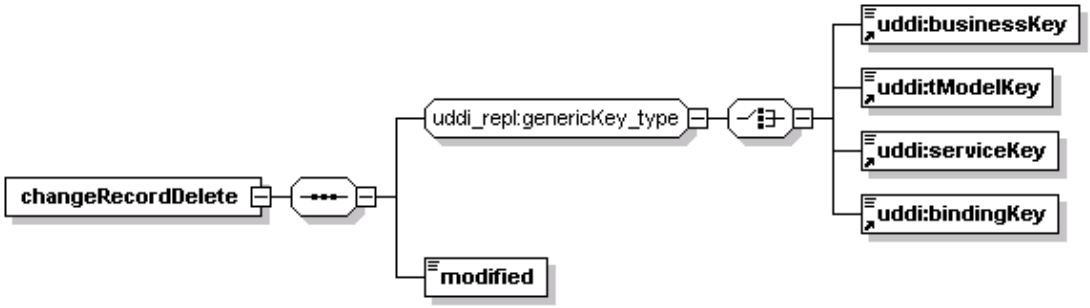


A changeRecordHide element corresponds to the behavior of hiding a tModel described in the delete_tModel in the Publish API section of this Specification. A tModel listed in a changeRecordHide should be marked as hidden, so that it is not returned in response to a find_tModel API call.

The changeRecordHide MUST contain a modified timestamp to allow multi-node registries to calculate consistent modifiedIncludingChildren timestamps as described in Section 3.8 *operationalInfo Structure*.

7.3.4 changeRecordDelete

The changeRecordDelete element is defined as follows:



A changeRecordDelete element indicates that an item defined in the UDDI registry is to no longer be used and expunged from the data stores in each of the nodes. The item to be deleted is indicated in the change record by the key of an appropriate entity type; this must contain the unique key of some businessEntity, businessService, bindingTemplate, or tModel that is presently defined. The changeRecordDelete element for deleting tModels corresponds to the administrative deletion of a tModel described in Section 6.1.3 *Updates and Deletions* of this specification. The changeRecordDelete for a tModel does not correspond to any API described in this specification and should only appear in the replication stream as the result of an administrative function to permanently remove a tModel.

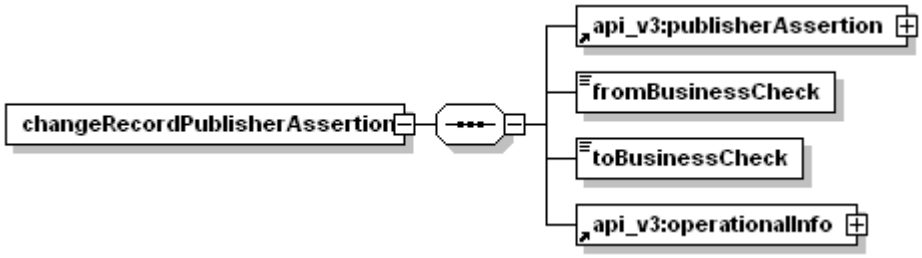
Permanent deletions of tModel information within the node may be made administratively. In this event, a UDDI Node may insert a delete operation into the replication stream. The publisher identifier for this operation is the account associated with the UDDI Node. Note that a permanent deletion of tModel information from the registry must have the prior approval of the other nodes participating within the registry.

The changeRecordDelete MUST contain a modified timestamp to allow multi-node registries to calculate consistent modifiedIncludingChildren timestamps as described in Section 3.8 *operationalInfo Structure*.

7.3.5 changeRecordPublisherAssertion

The changeRecordPublisherAssertion element describes the information that UDDI replication MUST convey in order to support the business-to-business relationship definition supported by UDDI.

An implementation MUST be able to determine the Registry changes from the information transmitted within the replication stream. The fromBusinessCheck and toBusinessCheck elements are Boolean values that represent which side of the business relationship is being asserted. A changeRecordPublisherAssertion message may include one or both sides of the relationship. For example, if the fromBusinessCheck is present and set to "true" then the assertion represents the parent-side of a parent-child relationship.



A changeRecordPublisherAssertion element indicates that one or both sides of the business relationship are to be inserted.

a. changeRecordPublisherAssertion with:

   <fromBusinessCheck>**true**</fromBusinessCheck> and <toBusinessCheck>**true**</toBusinessCheck> is used to indicate that both sides of the publisherAssertion (i.e., business relationship) are to be inserted. The two businessEntity elements that are referred to within the publisherAssertion MUST be in the custody of the node that originates the changeRecord.

b. changeRecordPublisherAssertion with:

   <fromBusinessCheck>**true**</fromBusinessCheck> and <toBusinessCheck>**false**</toBusinessCheck> is used to indicate that the fromBusinessCheck side of the publisherAssertion (i.e., business relationship) is to be inserted. The businessEntity that is referred to in the fromBusinessCheck MUST be in the custody of the node that originates the changeRecord.

c. changeRecordPublisherAssertion with:

   <fromBusinessCheck>**false**</fromBusinessCheck> and <toBusinessCheck>**true**</toBusinessCheck> is used to indicate that the toBusinessCheck side of the publisherAssertion (i.e., business relationship) is to be inserted. The businessEntity that is referred to in the toBusinessCheck MUST be in the custody of the node that originates the changeRecord.

d. changeRecordPublisherAssertion with:

   <fromBusinessCheck>**false**</fromBusinessCheck> and <toBusinessCheck>**false**</toBusinessCheck> if this is received in the replication stream, such a changeRecord will not generate any change to the registry. The node SHOULD log any events such as this.

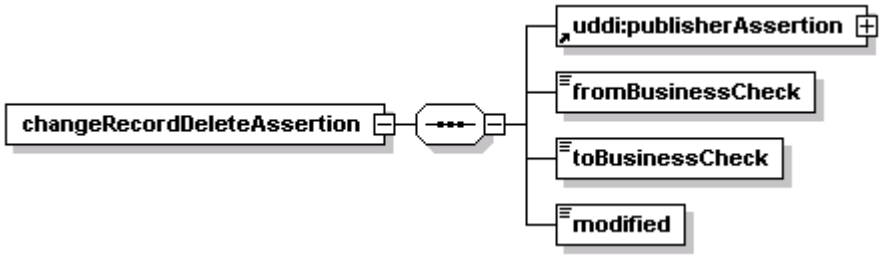
The operationalInfo element MUST contain a modified date corresponding to the update for the publisher assertion. This modified date should be stored by nodes supporting the subscription APIs in order to respond to subscription requests involving find_relatedBusinesses and get_assertionStatusReport filters. Since the publisherAssertions corresponding to a relationship may be originated from more than one node, the modified date stored for any relationship corresponding to the publisher should be the most recent date received from any node.

To handle signed publisherAssertion elements, it is necessary to indicate which set of signatures are being completely replaced as a result of the originating node's change to update one or both sides of the relationship represented by the publisherAssertion. The

optional signature element in the publisherAssertion must be ignored in replication and the toSignatures and fromSignatures elements must be used to replace signatures stored for the publisherAssertion. One of the elements toSignatures, fromSignatures or both must appear in the changeRecordPublisherAssertion. The presence of a toSignatures or fromSignatures element indicates that the signatures associated with the "to" or "from" side of the relationship must be deleted and completely replaced with the Signatures in the toSignatures or fromSignatures element. In the case where a single publisherAssertion represents both sides of the relationship, the node originating the corresponding changeRecordPublisherAssertion must include both a toSignatures and fromSignatures element with the identical set of Signature elements in both the toSignatures and fromSignatures. When the toSignatures element is not present, no changes are made to the signature elements associated with the "to" side of the relationship in the node. Similarily, when the fromSignatures element is not present, no changes are made to the signature elements associated with the "from" side of the relationship in the node.

7.3.6 changeRecordDeleteAssertion

The changeRecordDeleteAssertion element is defined as follows:



A changeRecordDeleteAssertion element indicates that one or both sides of the business relationship are to be deleted.

a. changeRecordDeleteAssertion with:

   <fromBusinessCheck>**true**</fromBusinessCheck> and <toBusinessCheck>**true**</toBusinessCheck> is used to indicate that both sides of the publisherAssertion (i.e., business relationship) are to be deleted. The two businessEntity elements that are referred to within the publisherAssertion MUST be in the custody of the node that originates the changeRecord.

b. changeRecordDeleteAssertion with:

   <fromBusinessCheck>**true**</fromBusinessCheck> and <toBusinessCheck>**false**</toBusinessCheck> is used to indicate that the fromBusinessCheck side of the publisherAssertion (i.e., business relationship) is to be deleted. The businessEntity that is referred to in the fromBusinessCheck MUST be in the custody of the node that originates the changeRecord.

c. changeRecordDeleteAssertion with:

   <fromBusinessCheck>**false**</fromBusinessCheck> and <toBusinessCheck>**true**</toBusinessCheck> is used to indicate that the toBusinessCheck side of the publisherAssertion (i.e., business relationship) is to be deleted. The businessEntity that is referred to in the toBusinessCheck MUST be in the custody of the node that originates the changeRecord.

d. changeRecordDeleteAssertion with:

   <fromBusinessCheck>**false**</fromBusinessCheck> and <toBusinessCheck>**false**</toBusinessCheck> if this is received in the replication stream,

such a changeRecord will not generate any change to the registry. The node SHOULD log any events such as this.

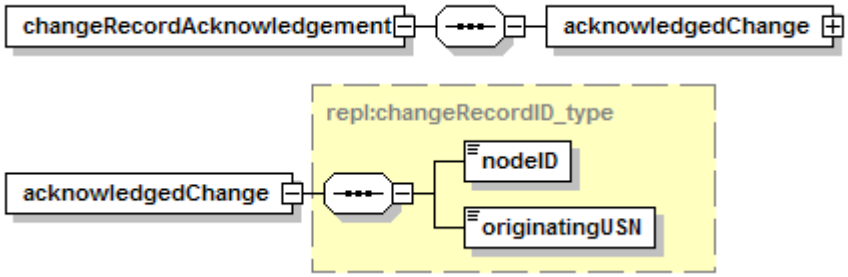
In the event that a businessEntity deleted with a delete_business API message references publisherAssertions, the node SHOULD NOT create corresponding changeRecords for the referenced publisherAssertions. Please refer to Section 7.2.6 *Replication Processing* for more discussion related to this.

To handle signed publisherAssertion elements, it is necessary to delete signatures corresponding to the deleted publisherAssertion. For each part of the relationship deleted as a result of this changeRecordDeleteAssertion, it is necessary to remove the all Signature elements associated with that part of the relationship.

The changeRecordDeleteAssertion element MUST contain a modified timestamp corresponding to the update for the publisher assertion. This modified date MUST be stored by nodes supporting the subscription APIs in order to respond to subscription requests involving find_relatedBusinesses and get_assertionStatusReport filters. Since the publisherAssertions corresponding to a relationship may be originated from more than one node, the modified timestamp stored for any relationship corresponding to the publisher MUST be the most recent date received from any node.

7.3.7 changeRecordAcknowledgment

The changeRecordAcknowledgement element is defined as follows:

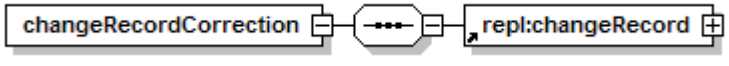


A node MUST originate a changeRecordAcknowledgement message when it receives and successfully processes a changeRecord that contains an acknowledgementRequested attribute set to true. The changeRecordAcknowledgement message contains the identification of the change that it is acknowledging.

It is specifically required that *all* nodes receiving a changeRecord with an acknowledgement request MUST originate an acknowledgement for it, even the node that originated the changeRecord in the first place.

7.3.8 changeRecordCorrection

A changeRecordCorrection contains information that is the corrected version of a change record that was previously originated in error. The correction simply contains the whole changeRecord that should have been transmitted in the first place; the originating node and originating USN information therein can be used to locate the offending record in change record journals.



When a node receives a changeRecordCorrection, it processes it only by making annotations in its change record journal; the data store of the node is *not* otherwise updated with the corrected change[29]. Specifically, and simply put, if the original offending change is still present in the node's change record journal, its entry SHOULD be merely annotated in such a way that if the change needs to be propagated to a replication partner that the correct contents of the change are transmitted instead of the original. The changeRecordCorrection may be considered successfully processed once this has been durably accomplished and the changeRecordCorrection itself durably recorded in the change record journal.

7.3.9 changeRecordNewDataConditional

From time to time, registries may find it useful to permit certain new data with certain particular keys to be introduced at possibly *any* of their nodes. For example, registries face the issue that the tModel representing the definition of a new *domainKey* partition in the keying scheme may be published at any node, and thus possibly at more than one node simultaneously. Some means is therefore necessary to prevent the introduction at multiple nodes of different (and thus conflicting) data under the key in question.

The changeRecordNewDataConditional element provides a means by which this can be accomplished. Such elements necessarily contain wholly *new* data, that is, data residing under a key that does not yet exist in the registry. The replication and processing of these records ensure that either (a) the new data contained therein is introduced throughout the registry without conflict with other such introductions, or (b) that a conflict is detected by the originating node in the registry and as a consequence informs all nodes the data not to be introduced. That is, race conditions are detected and resolved. Informing other nodes that the data should not be introduced is accomplished using changeRecordConditionFailed described in Section 7.3.10 *changeRecordConditionFailed*. When a node has determined that the data can be successfully introduced at all nodes, the originating node adds the changeRecordNewData element as another changeRecord in the change record journal.

It is a matter of registry policy which forms of data, if any, may be replicated within a given registry using the changeRecordNewDataConditional element.

Change records of type changeRecordNewDataConditional MUST be replicated in change records with an acknowledgementRequested attribute set to "true". For this type of change record nodes that originate such change records MAY emit their own acknowledgment to their own request at any time after having originated it.

If a Node A originates a change record *a* with payload of type changeRecordNewDataConditional, then, at the instant of origination (that is, at the instant at which the change record *a* is assigned its local USN by Node A), it MUST be the case at Node A that the following three rules all are adhered to:

1. **The new data is valid at Node A**. That is, Node A would at this instant be willing to have the data published into the registry's data set. Thus, the data in question must conform to all the specifications and policies applicable to it. In particular, for example, if the registry in question supports publisher assigned keys and the data is a tModel representing the introduction of a new domainKey, then the tModel MUST be categorized with the value "keyGenerator" from the uddi-org:types category system, must be trusted as legitimate according to the policy of the registry, and so on.

[29] The job of bringing data stores up to date should be addressed by a following change containing the now-current state of the data modified by the offending change.

2. **The key *k* is not an existing key at Node A**. That is, at Node A there does not exist a change record *x* with local USN less than that of the USN of the change record *a* where *x* contains a changeRecordNewData or changeRecordNewDataConditional payload using the key k in its datum, and there does not exist subsequent to x a change record z with payload changeRecordDelete or changeRecordNewDataConditional also containing the key *k*. In other words, there is no pre-existing data already using the proposed key, and Node A has not already issued a changeRecordNewDataConditional for key *k*.
3. **No other node is known by Node A to have requested the key *k* first**. That is, at Node A there does not exist a change record *x* with local USN less than that of the USN of the change record *a* where *x* contains a payload of type changeRecordNewDataConditional whose contained data also has key *k* and for which it has not been determined that *x* was involved in a race in the manner set forth below. In other words, Node A has not already acknowledged a changeRecordNewDataConditional which was issued by another node using the same key k, and which is still in a "conditional state".

7.3.9.1 Detection of collisions in conditional new data publication

The description within this section uses domain key generator tModels as an example. The behavior described MUST be applied to any type of new data that is found to collide with data from other nodes within the registry.

When two nodes have saved a domainKey key generator tModel with the same domainKey, a collision has occurred. Only one publisher can establish a domainKey domain at only one node. When multiple publishers attempt to establish ownership over a single key domain only one can be allowed to succeed to guarantee uniqueness of publisher assigned keys.

Suppose an arbitrary registry has three nodes A, B and C and publishers are attempting to save a tModel, T, with the same key at each node. As a result of the save_tModel Node A originates a changeRecordNewDataConditional change record a. SimilarlySimilarily, Node B also originates a changeRecordNewDataConditional change record (call it b) which contains a datum with same key as that of the datum in a. Finally, Node C be also originates a changeRecordNewDataConditional change record (call it c) which contains a datum with same key as that of the datum in a. Let the notation (x,Y) represent the acknowledgement by Node Y (in a changeRecordAcknowledgement originated by Node Y) of changeRecordNewDataConditional change record x, and let the notation Col(x) represent the changeRecordConditionFailed message for changeRecordNewDataConditional x..Then, recalling the ordering principle of replication of change records mentioned in Section 7.2.2 *Change Records*, all of the following scenario must be true:

Step	Operation	Node A	Node B	Node C
1	save T	**a**		
2	save T		**b**	
3	save T			**c**
4	A->B	**b**		
		(b, A)		
		Col(a)		
5	B->A		**a**	
			(a, B)	
			Col(b)	

6	A->C	**c** **(c, A)**		
7	B->C		**c** **(c, B)**	
8	A->B	**Col(b)** **(c, B)**		
9	B->A		**Col(a)** **(c, A)**	
10	C->A			**a** **(a, C)** **Col(c)** **b** **(b, C)** **(b, A)** **Col(a)** **(c, A)** **Col(b)** **(c, B)**
11	A->C	**(a, C)** **Col(c)** **(b, C)**		
12	B->C		**(a, C)** **Col(C)** **(b, C)**	

Please note in this scenario that nodes do not emit their own acknowledgement as they are allowed to and each node Y must generate (x, Y) when receiving x even if Y detects a collision. The originator of the changeRecordNewDataConditional must generate Col(x) when it detects a collision or must generate a changeRecordNewData after all the related acknowledgements have been received from the other nodes actively participating in the replication topology.

The operations at each node in the scenario above are:

1. T is saved on A that generates a, and inserts a in conditional storage.
2. T is saved on B that generates b, and inserts b in conditional storage.
3. T is saved on C that generates c, and inserts c in conditional storage.

4. A gets b from B by replication and detects a collision with a.
   A generates (b, A), and discards b and removes a from conditional storage[30].
   A also generates Col(a) indicating that a collision was detected and a will not be durably stored.
5. B gets a from A by replication and detects a collision with b.
   B generates (a, B), and discards a and removes b from conditional storage.
   B also generates Col(b) indicating that a collision was detected and b will not be durably stored.
   B also receives Col(a) which has already been discarded from conditional storage.
6. A gets c from C by replication. It is valid since there is no collision detected given that at step 4 evidence of the collision had been discarded.
   A generates (c, A), and A inserts c or (c, A)[31] into conditional storage (Rule 3).
7. B gets c from C by replication. It is valid since there is no collision detected given that at step 5 evidence of the collision had been discarded.
   B generates (c, B), and B inserts c or (c, B) into conditional storage (Rule 3).
8. A gets Col(b) and (c, B) from B by replication and can now consider T to be in a state where it can be successfully saved via to conditional record c (if Node A wanted to unnecessarily track such state information).
9. B gets Col(a) and (c, A) from A by replication and can now consider T to be in a state where it can be successfully saved via conditional record c (if Node B wanted to unnecessarily track such state information).
10. C gets a from A and detect a collision with c.
    C generates (a, C) and discards a and removes c from conditional storage.
    C also generates Col(c) indicating that a collision was detected and c will should not be durably stored anywhere.
    C also receives b from A and generates (b, C) and inserts b or (b, C) into conditional storage (Rule 3).
    C also receives (b, A) and does nothing (it is now waiting on a changeRecordNewData from B.)
    C also receives Col(a) from A, a or (a, C) are removed from conditional storage if they are still present.
    C also receives (c, A) for which c has already been discarded from conditional storage.
    C also receives Col(b) and discards b or (b, C) from conditional storage.
    C also receives (c, B) which has already been discarded from conditional storage.
11. A gets (a, C) from C and does nothing as a has already been discarded from conditional storage.
    A also gets Col(c) and discards c from conditional storage.
    A gets (b, C) from C and does nothing as b or (b, A) has already been discarded from conditional storage.
12. B gets (a, C) from C and does nothing as a or (a, B) has already been discarded from conditional storage.
    B also gets Col(c) and discards c or (c, B) from conditional storage.
    B gets (b, C) from C and does nothing as b has already been discarded from conditional storage.

[30] As an implementation detail, instead of removing b and a from conditional storage, the Node could remove a from conditional storage and wait to remove b until such time that a changeRecordConditionFailed change record is received.

[31] Whether c or (c, A) is inserted into conditional storage is an implementation issue. An implementation may choose to do either or both as a means of achieving the goals identified by Rule 3 i.e. requirement to prevent additional changeRecordNewDataConditional for a given T prior to having 1) receiving a changeRecordNewData or 2) a changeRecordConditionFailed.

Thus, at all nodes in the registry when Col(x) precedes any normal changeRecordNewData for x, none of the nodes will durably store x because the expected changeRecordNewData has been canceled by the changeRecordConditionalFailed.

Therefore, we can specify in a well-formed manner the following decision procedure:

- When a conditional record x is originated at any Node X, and when acknowledgement (x, xY) is seen for all nodes Y other than Node X before any other conditional record y is seen by Node X that conflicts with x, then Node X MUST conclude no such yx in fact exists, and that Node X can now safely publish the key in question, and that the data contained in the changeRecordNewData inside of record x can be successfully incorporated into the node's data store. Node X must also emit a changeRecordNewData for x to indicate that the data should be incorporated in the data store at all nodes.
- Conversely, if record yy is seen before all the acknowledgements from nodes other than Node X, then Node X SHOULD conclude that a race has occurred. Node X MUST then emit a changeRecordConditionFailed so that neither Node X nor any node that it was racing with has published the key, and that the introduction of neither records xx nor yy into the replication stream had any enduring effect on the data of the registry due to the changeRecordConditionFailed for x and any other records y that collided (the changeRecordNewDataConditional followed by the changeRecordConditionFailed for the same x is a no-opno-op).

In typical uses of the changeRecordNewDataConditional functionality, these sorts of races only arise due to the end-user error of attempting to simultaneously establish new data (such as *domainKey* key generators) at more than one node. The outcome of such errors is that the new data is not introduced at all, and thus the user must try again.

7.3.10 changeRecordConditionFailed

A changeRecordConditionFailed contains the changeID of a conditional change record that has failed to meet the criteria established by the registry for insertion into the data model at each node. The changeID identified the changeRecord that MUST be removed from any conditional storage; the originating node and originating USN information therein can be used to locate the changeRecordNewDataConditional record in change record journals.

7.4 Replication API Set

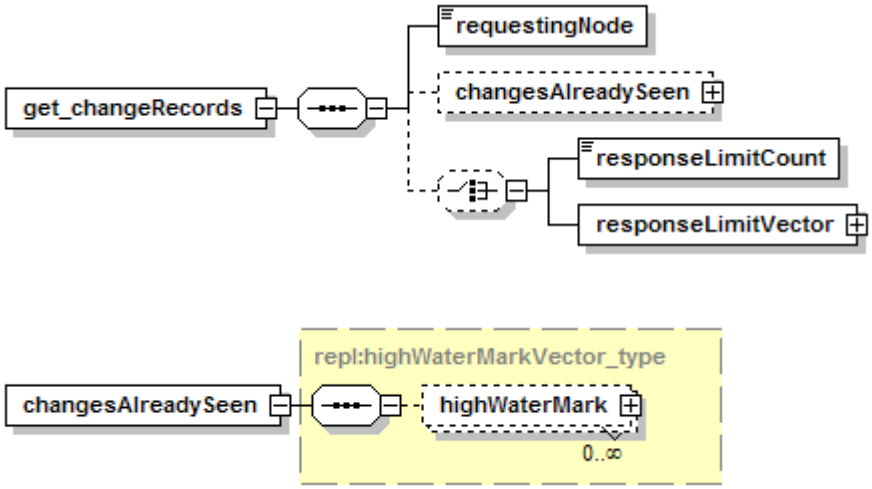
UDDI Replication defines four APIs. The first two presented here are used to perform replication and issue notifications. The latter ancillary APIs provide support for other aspects of UDDI Replication.

- get_changeRecords
- notify_changeRecordsAvailable
- do_ping
- get_highWaterMarks

7.4.1 get_changeRecords Message

The get_changeRecords message is used to initiate the replication of change records from one node to another. The caller, who wishes to receive new change records, provides as part of the message a high water mark vector. This is used by the replication source node to determine what change records satisfy the caller's request.

7.4.1.1 Syntax



7.4.1.2 Arguments

- ***requestingNode***: The requestingNode element provides the identity of the calling node. This is the unique key for the calling node and SHOULD be specified within the Replication Configuration Structure.
- ***changesAlreadySeen***: The changesAlreadySeen element, if present, indicates changes from each node that the requestor has successfully processed, and thus which should not be resent, if possible.
- ***responseLimitCount*** or ***responseLimitVector***: A caller MAY place an upper bound on the number of change records he wishes to receive in response to this message by either providing a integer responseLimitCount, or, using responseLimitVector, indicating for each node in the graph the first change originating there that he does *not* wish to be returned.

More specifically, the recipient determines the particular change records that are returned by comparing the originating USNs in the caller's high water mark vector with the originating USNs of each of the changes the recipient has seen from others or generated by itself. The recipient SHOULD only return change records that have originating USNs that are greater than those listed in the changesAlreadySeen highWaterMarkVector and less than the limit required by either the responseLimitCount or the responseLimitVector.

In nodes that support pre-bundled replication responses, the recipient of the get_changeRecords message MAY return more change records than requested by the caller. In this scenario, the caller MUST also be prepared to deal with such redundant changes where a USN is less than the USN specified in the changesAlreadySeen highWaterMarkVector.

The response to a get_changeRecords message is a changeRecords element. Under all circumstances, all change records returned therein by the message recipient MUST be returned sorted in increasing order according to the recipient's local USN.

7.4.1.3 Returns:

A node will respond with the corresponding changeRecords.

7.4.1.4 Caveats:

Processing an inbound replication message may fail due to a server internal error. The common behavior for all error cases is to return an E_fatalError error code. Error reporting SHALL be that specified by Section 4.8 – *Success and Error Reporting* of this specification.

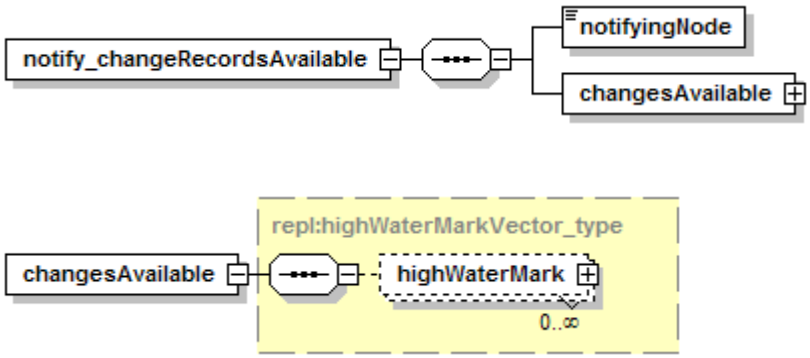
7.4.2 notify_changeRecordsAvailable Message

Nodes can inform other nodes that they have new change records available for consumption by replication by using this message. This provides a proactive means through which replication can be initiated, potentially reducing the latency of the dissemination of changes throughout the set of UDDI nodes. The notify_changeRecordsAvailable message is the predecessor to the get_changeRecords message.

Each node MUST respond with the message defined within the Section 7.4.2.3 *Returns* when a valid notify_changeRecordsAvailable message is received.

At an interval set by policy after the origination of new change records within its node, a node SHOULD send this message to each of the other nodes with which it is configured to communicate this message according to the currently configured communication graph. It SHOULD ignore any response (errors or otherwise) returned by such invocations.

7.4.2.1 Syntax



7.4.2.2 Arguments

- ***notifyingNode***: The parameter to this message indicates that the notifyingNode has available the indicated set of changes for request via get_changeRecords.
- ***changesAvailable***: When sending the notify_changeRecordsAvailable message, a node shall provide a high water mark vector identifying what changes it knows to exist both locally and on other nodes with which it might have had communications. Typically, no communication graph restrictions are present for the notify_changeRecordsAvailable message. In the event that the sending node does not know the USN for a specific node within the CommunicationGraph, the changesAvailable element MAY contain a highWaterMark for that node with an unspecified nodeID element.

7.4.2.3 Returns

Success reporting SHALL be that specified by Section 4.8 – *Success and Error Reporting* of this specification.

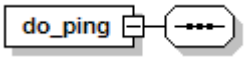
7.4.2.4 Caveats:

Processing an inbound replication message may fail due to a server internal error. The common behavior for all error cases is to return an E_fatalError error code. Error reporting SHALL be that specified by Section 4.8 – *Success and Error Reporting* of this specification.

7.4.3 do_ping Message

This UDDI API message provides the means by which the current existence and replication readiness of a node may be obtained.

7.4.3.1 Syntax



7.4.3.2 Arguments

None

7.4.3.3 Returns

The response to this message must contain the operatorNodeID element of the pinged node.

7.4.3.4 Caveats:

Processing an inbound replication message may fail due to a server internal error. The common behavior for all error cases is to return an E_fatalError error code. Error reporting SHALL be that specified by Section 4.8 – *Success and Error Reporting* of this specification.

7.4.4 get_highWaterMarks Message

This UDDI API message provides a means to obtain a list of highWaterMark element containing the highest known USN for all nodes in the replication graph.

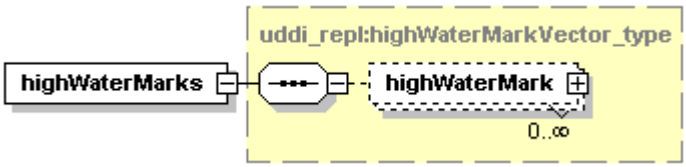
7.4.4.1 Syntax

7.4.4.2 Arguments

None

7.4.4.3 Returns

A highWaterMarks element is returned that contains a list of highWaterMark elements listing the highest known USN for all nodes in the replication communication graph. See Section 7.2.4 *High Water Mark Vector* for details.



If the highest originatingUSN for a specific node within the registry is not known, then the responding node MUST return a highWaterMark for that node with an originatingUSN of 0 (zero).

```
<highWaterMark>
   <nodeID>…</nodeID>
   <originatingUSN>0</originatingUSN>
</highWaterMark>
```

7.4.4.4 Caveats:

Processing an inbound replication message may fail due to a server internal error. The common behavior for all error cases is to return an E_fatalError error code. Error reporting SHALL be that specified by Section 4.8 – *Success and Error Reporting* of this specification.

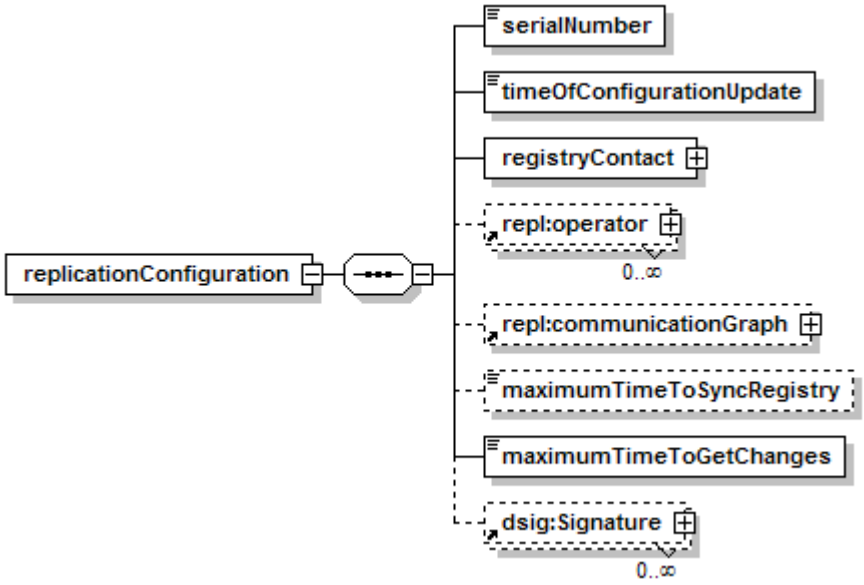
7.5 Replication Configuration

7.5.1 Replication Configuration Structure

The replication of UDDI data within a registry SHOULD be governed by information maintained within a Replication Configuration Structure. This structure MAY be contained within a *Replication Configuration File*. The structure includes sufficient information to uniquely identify each node within the UDDI registry. If used, each node within the registry MUST specify at least one contact as described below.

If used, UDDI nodes MUST manage any and all changes to the contents of the Replication Configuration Structure.

The XML schema definition describing the replicationConfiguration element is shown below. The root element of an instance document of this XML schema must be a replicationConfiguration element as defined in the UDDI v3 replication schema[32]:



The Replication Configuration Structure contains a serial number which is increased by at least one each time the published configuration is updated or changed. For the convenience of users, the element timeOfConfigurationUpdate identifies the time of the last update. The formatting of the timeOfConfigurationUpdate element is described later in this specification. The registryContact identifies a party who maintains and updates the Replication Configuration Structure.

The element, maximumTimeToGetChanges, allows for the specification of the maximum amount of time (in hours) that an individual node may wait to request changes. Nodes MUST perform a get_changeRecords replication message within the time frame defined by the value

[32] All schema definitions here are presented per the 2001 Recommendation of XML Schema. Note that the schema definitions in this present document should be considered as informative only; the normative schema definitions are found in an accompanying .XSD specification file.

of the maximumTimeToGetChanges element defined within the Replication Configuration Structure. Thus, change data can always be propagated throughout the UDDI registry within a finite amount of time, while at the same time changes will often propagate quickly. Use of this element is determined by registry policy as detailed in Section 9.6.4 *Replication Policies*.

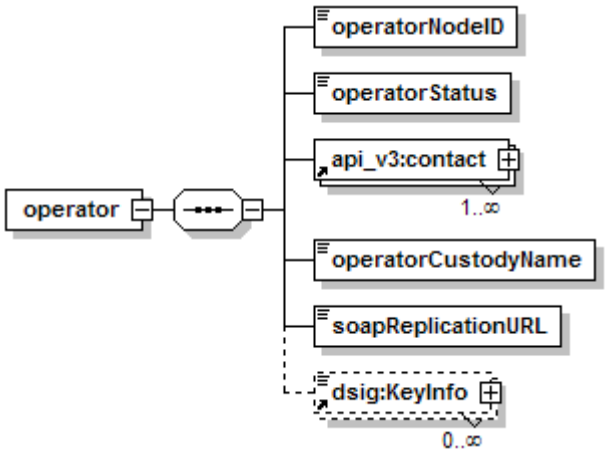
The operator elements in a Replication Configuration Structure are a list of the nodes and established paths of communication between the nodes within a registry. The communication paths and general replication topology considerations are discussed later in this specification.

The element, maximumTimeToSyncRegistry, allows for the specification of when (in hours) a change made at any single node in the Registry is expected to be visible at all nodes within the registry. The element, maximumTimeToGetChanges, allows for the specification of the maximum amount of time (in hours) that an individual node may wait to request changes. Use of this element is determined by registry policy as detailed in Section 9.6.4 *Replication Policies*.

The dsig:Signature elements in a Replication Configuration Structure allow for signing of the document for assurance of its integrity using XML Digital Signature.

7.5.2 Configuration of a UDDI Node – operator element

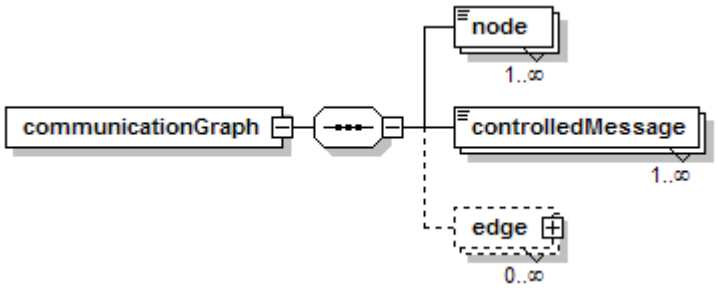
Each current UDDI node within the Registry is identified with an operator element in the replicationConfiguration:

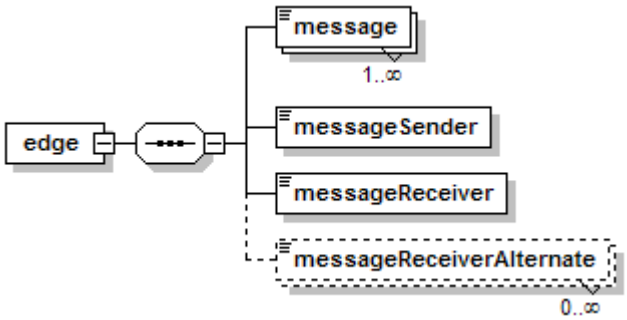


The operatorNodeID contains a unique key that is used to uniquely identify this node throughout the UDDI registry. The value used MUST match the businessKey of the Node Business Entity as referenced in Section 6.2.2 *Self-Registration of Node Business Entity*. The contact or contacts listed provide information about humans who should be contacted in the face of administrative and technical situations of various sorts. . The dsig:KeyInfo elements are intended to contain the certificate details if the soapReplicationURL makes use of Secure Sockets Layer 3.0 with mutual authentication as described in Section 7.5.5 *Security Configuration*.

7.5.3 Replication Communication Graph

The Replication Configuration Structure provides a means by which the UDDI replication traffic can be controlled and administered. This is achieved with the use of a communicationGraph element:





The communicationGraph element begins by explicitly listing a unique ID for each node within the registry. This second listing of the unique IDs within the Replication Configuration Structure enables the separate control of communication with a node from the management of other aspects of the node's configuration.

Following the listing of nodes is the controlledMessage element that lists the set of messages over which this communication graph is intended to administer control of. If a message element local name is listed in the controlledMessage element, then such messages SHALL only be sent between nodes that are listed in the subsequent edges of the graph. In contrast, communication restrictions are not imposed on replication messages not identified in the controlledMessage element.

The next element of the communication graph, the edge element, identifies the graph edges imposed for those nodes that are listed. All node ID's listed in an edge MUST have been previously identified in the node element of the communication graph.

Each edge in the graph is a directed edge. The message elements contain the local name of the Replication API message elements. They indicate that only messages of the type explicitly identified for a particular edge MAY be sent from the specified messageSender to the specified messageReceiver. Restricted two-way communication between nodes MUST, if desired, be identified as a pair of edges with opposing directionality. A given directed edge MAY be listed at most once: for each edge, the pairing (messageSender, messageReceiver) MUST be unique over the entire set of edges of the graph.

For each directed edge, an ordered sequence of zero or more alternate, backup edges MAY be listed using the messageReceiverAlternate element. Should a communications failure prevent message communication over the indicated primary edge, the backup edges MAY be tried in order until communication succeeds.

In the absence of a communicationGraph element from the Replication Configuration Structure, all nodes listed in the node element MAY send any and all messages to any other node of the registry.

A non-normative example of the use of the CommunicationGraph can be found within Appendix J *UDDI Replication Examples.*

7.5.4 SOAP Configuration

In UDDI, node-to-node replication communication MUST be carried out by means of SOAP messages and responses. The soapReplicationURL element of the operator element indicates where such messages should be sent to communicate with a given node. Specifically, in order to carry out a message invocation of type X with a given node in the registry, a message is sent using HTTP POST, as described in Chapter 4, to the URL identified within the Replication Configuration Structure.

7.5.5 Security Configuration

Mutual authentication of UDDI nodes is RECOMMENDED. This MAY be achieved using mutual X.509v3 certificate-based authentication as described in the Secure Sockets Layer (SSL) 3.0 protocol. SSL 3.0 with mutual authentication is represented by the tModel uddi-org:mutualAuthenticatedSSL3 as described within Section 11.3.2 *Secure Sockets Layer Version 3 with Mutual Authentication*. The certificate credentials that SHOULD be used for the mutual authentication SHOULD be included in the dsig:KeyInfo element(s) for each node Replication Configuration Structure.

7.6 Error Detection and Processing

While replication errors are expected to happen rarely, mechanisms must be in place to detect, deal with, and help manage the correction of such problems.

Throughout this section we will consider a UDDI registry consisting of 4 nodes (A, B, C, and D) configured in a cyclic replication pattern such that D places get_changeRecords requests to C (D>C), C>B, B>A, A>D. In addition to each node's primary replication edge a set of alternate edges have been defined for each node which, in case of difficulty, permits it to bypass each upstream node in turn (i.e. B has two secondary edges: B>D and B>C). For clarity, only one is depicted below in Figure 4.

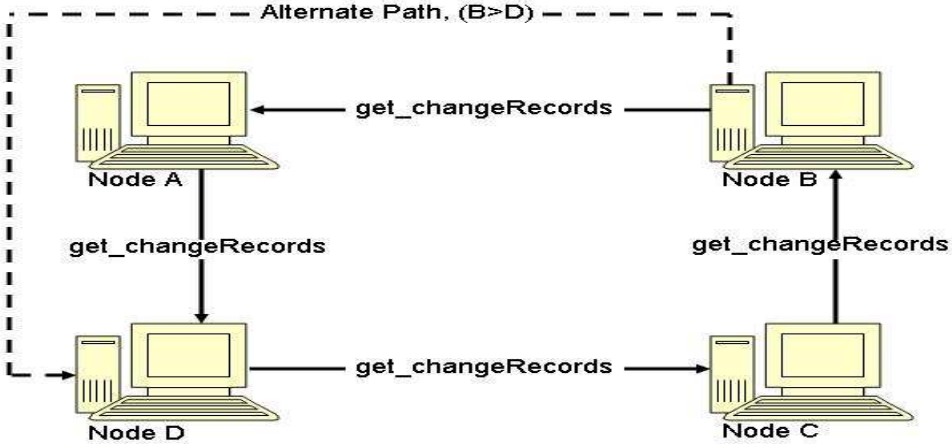


Figure 4 - Alternate Edge Example

Asserting an invalid change record, it follows that:

In all of the cases below we assume that the following steps are executed, in the order listed:

1. A Publisher of data held in custody at Node D saves changes to that data
2. Node D originates a change record x.
3. Node A issues a get_changeRecords request to Node D and successfully processes the change record set containing x which D issued in response.
4. Node B issues a get_changeRecords request to Node A and processes the change record set that Node A issued in response. During processing of the change record set Node B encounters x and is unable to validate it.

When Node B is unable to validate x it is required to signal that it has encountered a change record that it considers invalid (see Section 7.7 *Validation of Replicated Data* for a precise definition of what it means for a change record to be 'invalid'). It does this by communicating to all of the UDDI nodes. The communication must include the following:

- The originating Node's USN Value
- The reporting Operator Node ID
- The Originating Node ID
- The Replication operation type being processed (i.e., changeRecordPayload_type)
- The unique key of the datum being processed
- The type of datum being processed (i.e., businessEntity, businessService, tModel, etc.)

In addition Node B must refuse to accept *x* from Node A (i.e. it MUST NOT add x to its journal, MUST NOT process x into its registry, and MUST NOT update its high water mark vector to reflect processing of x) and MUST NOT consider *x* to be successfully processed. Recall that Node A is required to deliver records ordered by local USN and Node B is required to consume records in the order they are provided. As a consequence, Node B will for the moment be unable to accept any *other* new records from Node A. Note that Node B may have successfully processed other change records from the change record set it received from Node A. If those change records preceded x, then they are unaffected by the issues associated with x.

This design avoids polluting Node B with data that it knows to be invalid, while preserving the important ordering guarantee of change propagation upon which other replication semantics rely.

7.6.1 UDDI Registry Investigation and Correction

Once Node B has asserted the existence of an invalid change record the UDDI nodes MUST take remedial action. The investigation of Node B's assertion regarding x MAY have a number of outcomes. Those are documented in the cases below.

The following change records are used in these cases:

- x: a changeRecordNewData message created by Node D that Node B asserts is invalid.
- y: a changeRecordCorrection message created by Node D containing x', a corrected payload for x.

- z: a changeRecordNewData message created by Node D containing current payload for x.

7.6.1.1 Case 1: Invalid record validation

The UDDI nodes MAY determine that Node B's assertion that x is invalid MAY be a result of overly aggressive checking and validation in Node B's UDDI implementation. In this case:

- Node B WILL BE REQUIRED to change its code,
- Node B WILL continue to participate in replication cycles, retrying propagation of x from A to B in each cycle until the code changes are deployed and x is processed successfully.

7.6.1.2 Case 2: Invalid interim representation

The assertion by Node B that x is incorrect MAY reflect incorrect handling of x by interim Node A (recall that x is replicated first from node D to Node A, then from Node A to Node B). Note that the originating Node D contains a valid x. In these cases:

- Node A WILL BE REQUIRED to change its code.
- Node D WILL BE REQUIRED to generate y and z.
- Node A WILL continue to participate in replication, processing y (annotating x in its journal to refer to x' in y) and z in one or more change record sets from Node D,
- Node B WILL continue to participate in replication in subsequent cycles. In each it WILL attempt to propagate x from Node A, then begin to exercise its alternate edges, descending one additional edge each cycle (i.e. in cycle (n+1) it WILL attempt to replicate using B>A and B>D). At some point node B MAY process a valid x from an alternate node, in these instances, Node B WILL later process y (annotating x to point to x' in y) and z from Node A. If Node B has not processed a valid x from an alternate node prior to issuing a get_changeRecords to Node A which has processed y, then Node B WILL process x', y (annotating x to point to x' in y), and z from Node A.
- Node C WILL continue replication and will receive x, y, and z from Node B.

Expanded Description:

Handling of the assertion that change record x delivered by Node A is invalid begins in replication cycle (n+0) with Node B communicating the details of the issue to all nodes within the UDDI registry, discarding x and all subsequent change records received from Node A, and halting its replication.

During its next replication cycle (n+1) Node B WILL attempt to process a change record set from Node A. Assuming that Node A has not yet processed y produced by Node D, Node B WILL encounter an invalid x as the first record in the change record. In this case Node B WILL NOT issue a redundant message including the details of the issue to all nodes within the UDDI registry, however it WILL again discard x and all subsequent change records within the set issued by Node A. Node B WILL then use its first alternate replication edge (B>D) to circumvent Node A and will process a change record set containing x returned by Node D.

In its next replication cycle (n+2) Node B WILL successfully process a change record returned by Node A. Since Node B's high water mark vector entry for Node D has been updated to reflect its successful processing of x, Node A WILL NOT be asked to deliver x, and should it do so Node B WILL NOT attempt to process it (recall that a node responding to a get_changeRecords request MUST reply with at least the data requested, but MAY respond with more data than was requested by the caller).

During an arbitrary replication cycle following Node B's receipt of an invalid x from Node A, Node D WILL produce a changeRecordCorrection y containing a valid version x' of x. Node A

WILL process y annotating x within its journal to indicate that the payload in x is replaced by x' in y. Note that although Node A's journal now contains a valid x, Node A's registry does not.

Node B's next get_changeRecords request to Node A WILL result in a change record set containing either x' and y or only y. Change record x' is delivered in instances where Node A processed y before Node B issued its (n+1) get_changeRecords request to A.

Following production of y, Node D WILL produce z (which contains the now-current contents of the information manipulated by x). Change record z WILL be propagated normally through replication.

7.6.1.3 Case 3: Invalid generation

The assertion that x is invalid MAY be due to Node D having improperly generated x. In this case:

- Node D WILL BE REQUIRED to change its code so that in future it generates correct change records, and to generate y and z.
- Node A WILL BE REQUIRED to change its code so that in future it detects invalid change records such as x.
- Node A WILL continue participating in replication, processing y (annotating x to point to x' in y) and z.
- Node B WILL continue participating in cycles, continuing to attempt propagation of *x* from Node A to Node B until it processes x', y (annotating x to point to x' in y), and z.
- Node C WILL continue replication cycles, receiving x', y (annotating x to point to x' in y), and z.

Expanded Description:

In this case handling of the assertion that record x delivered by Node A is invalid begins in replication cycle (n+0) with Node B communicating the details of the issue to all nodes within the UDDI registry, discarding x and all subsequent change records received from Node A, and halting its replication.

During its next replication cycle (n+1) Node B WILL again process a change record set from Node A. It should encounter x as the first record in the change record set returned by Node A. If processing of x again yields an error, Node B WILL NOT issue a redundant message including the details of the issue to all nodes within the UDDI registry, it WILL discard x and all subsequent change records from Node A. Node B WILL then use its first alternate replication edge (B>D) to circumvent Node A and request change records from another node (D). In processing the change record set returned by Node D it WILL encounter an invalid x. It WILL then communicate the details of the issue to all nodes within the UDDI registry (pointing to Node D as the originating and the sending node), discard x and all subsequent change records received from Node D, and halt its replication.

In its next replication cycle (n+2) Node B WILL again process a change record set from Node A. Should it again encounter an invalid x as the first record in the set it WILL discard x and all subsequent change records from Node A and proceed to its first alternate replication (B>D). Should processing a change record set from Node D again uncover an invalid x as the first record in the set from Node D, Node B WILL discard x and all subsequent change records from Node D and proceed to its second alternate replication edge (B>C) to circumvent nodes A and D. Since Node C has not received x the change record set is produces should be valid.

During an arbitrary replication cycle following production of x, Node D WILL produce a changeRecordCorrection y containing the corrected version x' of x. This new change record y propagates using the normal replication mechanism. Node A WILL process y, annotating x in

its journal to now refer to x' in y. At this point Node A's journal is correct, however its registry still contains an invalid version of x.

In a subsequent replication cycle, Nodes B and C WILL process x' and y. Following production of y, Node D WILL produce z. In a subsequent replication cycle Nodes A, B, and C WILL process z. At this point all three nodes contain a valid representation of the data addressed by x.

7.7 Validation of Replicated Data

UDDI nodes MUST enforce the validity of all the data entering their data store, both data they originate themselves, and data that they receive from other nodes through replication. A registry policy should define the level of validation that nodes enforce above and beyond validating that the data conforms to the UDDI schemas and does not cause corruption of the node data store.

Specifically, it is required that a node fail to accept any publication API request which would, were it accepted, at that instant put its data store into an invalid state. Nodes must also support analogous enforcement through any user interfaces or other means by which the data in their data store may be added to or updated.

Moreover, as a node receives replicated changeRecords from another node in response to a get_changeRecords request, it must consider the potential effect of the incorporation of the change into its data store. If the incorporation by the receiving node of such a change record (together, of course, with any preceding changes) would put its data store into an invalid state, then an error in one or more of the UDDI node implementations within the registry has been detected. In response, the receiving node must carry out the error detection processing sequence described in Section 7.6 *Error Detection and Processing*.

Upon the receipt of changeRecords related to publisherAssertions that refer to businesses that have been previously deleted, or access point information that refers to invalid bindingKeys, or a tModelKey of a keyedReference that refers to a tModel that no longer exist, or any attempts to project a service that no longer exist at the node, or a reference in a isReplacedBy keyedReference to an invalid or missing businessEntity or tModel key, nodes MUST NOT raise replication errors. Nodes MUST include the respective changeRecords in a response to relevant get_changeRecord messages.

Note as a point of implementation that it might often be useful to batch together several incoming change records under one validity check; this is a valid optimization since later change records cannot adversely affect the validity of earlier changes. However, should such an optimized validity check fail, an implementation must be prepared to back out of and fail to accept the entire set of candidate change records involved and then reconsider each individually in turn.

As a checked value set reference may be validated or checked at an originating node within the registry, it need not be checked again at a node in the midst of processing a replication stream. If a checked value set reference is checked again during replication it SHOULD not generate a replication stream error.

7.8 Adding a Node to a Registry Using Replication

The following steps MAY be used to add a node to a registry that uses UDDI v3 replication. A registry policy may be defined for the addition of a node and the policy MAY define a different or more detailed process than the following process:

1. Add the new node to the Replication Configuration Structure with the operatorStatus value set to "new". If a communicationGraph is used in the registry's replication Configuration Structure, one or more edges MUST be added so the new node may call the get_changeRecords API.
2. Administrators of the existing nodes are notified of the update. They each retrieve and process the new information in the configuration file in order to add the new node as one of the parties with which their implementations are willing to authenticate and communicate. To verify the ability to communicate, each node pair within the current UDDI registry SHALL successfully exchange the do_ping message.

3. Once all existing nodes within the registry have verified the ability to successfully communicate with the new node, the configuration file is changed a second time to add appropriate edges to the communicationGraph in order to introduce the new node into the active replication graph. The messages allowed between the new node and its primary may be restricted to allow for startup processing. Replication then proceeds to provide change records to the new node which establish in it a current image of the registry.
4. The new node MUST issue a get_changeRecords API call to an existing node and SHOULD process all available changeRecord elements. As nodes MAY limit the number of changeRecord elements in a response, processing all available changeRecord elements MAY require several get_changeRecords API calls.
5. When the new node has completed processing of the change history and is ready to engage in all UDDI registry activities, the operatorStatus value, within the Replication Configuration Structure, WILL be updated from "new" to "normal" and any message restriction that may have been imposed earlier may be removed. The updated replicationConfiguration will be distributed to the nodes defined within the communicationGraph via replication.

7.9 Removing a Node from a Registry Using Replication

The following steps MAY be used to remove a node to a registry that uses UDDI v3 replication. A registry policy may be defined for the removal of a node and the policy MAY define a different or more detailed process than the following process

1. At least one node in the registry MUST obtain all changeRecord elements representing the final state of the node that will be removed.
2. The value of the operatorStatus element in the operator element in the Replication Configuration Structure of the node that is being removed MUST be changed from "normal" to "resigned." In addition any edge elements that include the node that is being removed MUST be removed from the communicationGraph element. The operator element for the removed node remains in the Replication Configuration Structure so other nodes in the registry accept the changeRecord elements from the removed node.
3. Depending on registry policy, the custody of the UDDI data from the removed node MAY subsequently be changed to one or more nodes in the registry.

8 Publishing Across Multiple Registries

In prior versions of the UDDI Specification, the behavior in which a publisher wished to copy the entirety of a UDDI registry entity from one registry to another while preserving the identical key was explicitly not allowed. The Version 1 and 2 specifications mandated that *only* the UDDI node could generate keys. A publisher could not pre-assign the key of a UDDI entity. With this stipulation in place, a publisher could not import or export data between registries. The rationale behind this mandate was to insure that no duplicate keys would ever be generated in a given registry because only nodes within that registry had the authority to generate keys. Consequently data sharing between registries in Version 1 and 2 of UDDI was functionally not allowed.

Version 3 of UDDI approaches the issue of key generation in a significantly different fashion and, as such, the possibility of publishing an entity to another UDDI registry while preserving the key is allowed. This behavior is known as *entity promotion*. With this version of UDDI, a publisher is permitted to propose a new key for an entity, and, given the policies of a registry, that key and the entity associated with that key may be inserted into the registry. Thus, the possibility of sharing data among UDDI registries is a reality and, with this new functionality, UDDI's scope in terms of a more broadly distributed environment is made manifest.

At its core, registry-to-registry data sharing, or the publishing of data across multiple registries, involves the sharing of data between two UDDI registries. With such a notion, individual UDDI registries can be connected in complex ways. In fact, for a given registry, a publisher may have the authority to add many entities to a registry and, as such, a publisher could potentially publish the entirety of a registry's contents into another registry, effectively mirroring the data. Or, the publisher might be interested in only a subset of data from another registry and only copy a portion of that data. When considering the implications of entity promotion, it is important to conceive of a publisher in a broader context, which makes possible a number of interesting scenarios.

The most important aspect to consider in terms of data that has been published across multiple registries is that no a priori assumptions can be made about the nature of a relationship between the data in two registries and, by implication, no assumption can be made about the relationship between an entity that happens to exist in two (or more) registries. The data associated with a given key is by default scoped to a single registry and not to multiple registries. Consequently, inter-*registry* publication is quite different than inter-*node* behavior, i.e., the behavior of replication of nodes within a single registry. In replication, there is a normative model for the nodes that participate in a replication topology and the for data management of those nodes. However, a normative model for *inter*-registry publication does not exist and therefore cannot be assumed. The nature of interaction between two given registries is not mandated by the UDDI specification. The trust models and data management procedures between two registries are a matter of policy established by the two registries and are defined as non-normative behavior in terms of the UDDI v3 specification. In this way, inter-*registry* interaction differs significantly from inter-*node* replication.

While normative inter-registry behavior is outside the scope of the specification, an explanation of exactly *how* the specification enables the ability to share data between registries is in order. The focus of this chapter is to outline further the rationale for inter-registry communication, to introduce the different kinds of inter-registry communication, and to provide guidance when considering engaging in inter-registry publication.

8.1 Relationships between Registries

8.1.1 Root Registries and Affiliate Registries

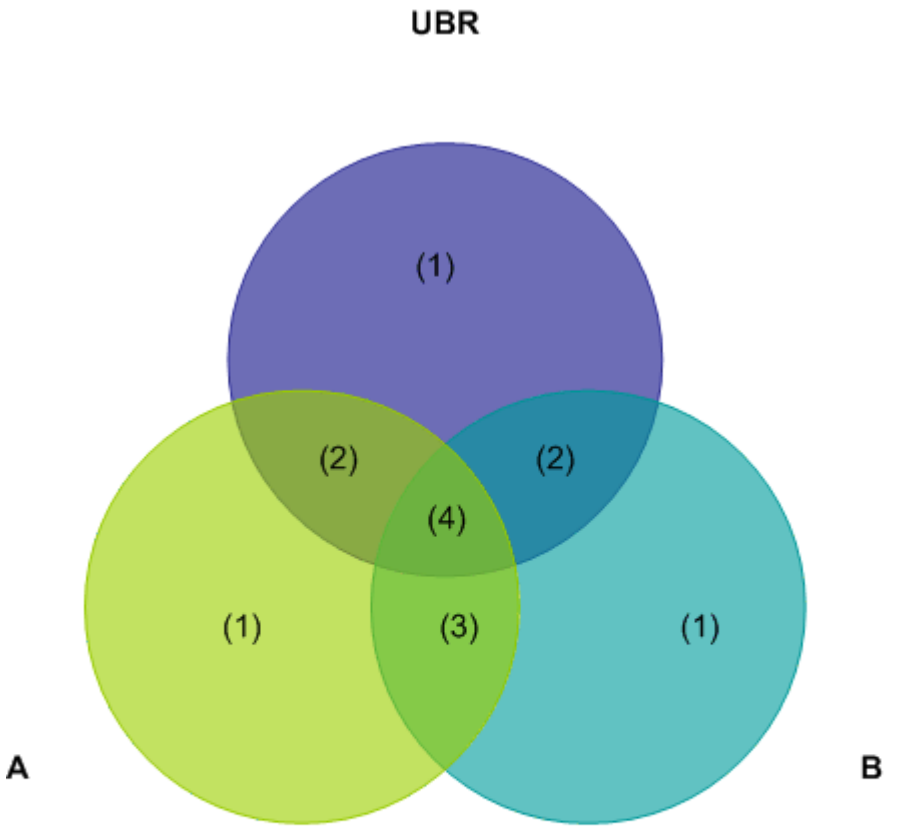
For this new capability to be useful, care must be taken to avoid key collision between registries. The recommended way to prevent such collision is to establish a *root registry*, a registry that acts as the authority for key spaces. A root registry serves to delegate key partitions such that other registries can rely upon the root registry for verification and validation of a given key partition. All other registries that interact with the root registry are called *affiliate* registries. Affiliate registries rely on the root registry to delegate key partitions and insure that uniqueness across key partitions is maintained. By relying on a common root registry as an arbitrator of key spaces, affiliate registries can share data with both the root registry and among one another with the knowledge that a given partition is unique. Note that it is still the responsibility of each registry, both a root registry and its affiliates, to insure the data integrity and uniqueness of the keys within its custody.

An important example of a root registry is the UDDI Business Registry, which has a set of policies in place to generate unique uuidKeys as well as to validate domainKeys through signatures that correlate with DNS records. These policies insure the uniqueness of both domainKeys and uuidKeys within the UDDI Business Registry, and thus the UBR serves as a reasonable root registry for many purposes. In fact, establishing alternate root registries is not recommended, as this would ultimately defeat the goal of publishers being able to share data between multiple registries with an assurance of avoiding a key collision. By acknowledging the UDDI Business Registry as a root, an affiliate registry can establish inter-registry communication policies and procedures with both the UDDI Business Registry and any other registry which is an affiliate of the UDDI Business Registry.

8.1.2 A Closer Look at Inter-Registry Communication Models

Before looking at the operational details of sharing data between registries, a look at some of the permutations of inter-registry communication is in order. Using a Venn diagram to represent the cross publication models between three registries sheds some light on permutations of data sharing. Note that the diagram below does not represent the directional flow with regard to how exactly how the data might move between registries. However, the diagram does provide a conceptual framework to begin thinking about how the data in multiple registries might be related.

In the Venn diagram below, the circle (1) represents the UDDI Business Registry, a root registry, while the two other circles represent registry A and registry B, two affiliate registries



This diagram demonstrates the different kinds of data sharing that can occur among 3 registries: the UBR, affiliate A and affiliate B. There are cases (1) when the data in each registry exists independently, (2) when data exists in both an affiliate and the root registry, (3) when data exists in two affiliates but not in the root registry and (4) when data exists in all three registries.

Consider Scenario (1), in which the three registries contain data that exist independent of the other registries. There are certainly scenarios that exist in which data would not be of value or interest to another registry, and thus would exist in one and only one registry. In this case, it may appear that there are not concerns about possible key collision with another registry.

However, it is critical to realize that, registry policy aside, *nothing prevents a user of a registry from promoting entities from one registry to another registry*. Put another way, no registry is an island. A user with read authority in one registry and publish authority in some other registry can retrieve entities from the first and publish them in the second. And, once that data has been promoted, nothing prevents the publisher from altering the data in the original registry. This concern is mitigated with the advent of digital signature support. A signed UDDI entity can be copied across multiple registries and the publisher can verify that the original content has not been modified.

It is important for both nodes and clients to remember that a registry typically does not control the data outside of itself. A client of a given registry must be aware that different registries might contain identical keys with different data. In some scenarios, this is the desired

behavior, but there are also cases where such behavior could be construed as malicious. As such, clients need to be cognizant of the policies and procedures of registries they interact with.

In Scenario (2), Affiliate A and Affiliate B each have data that exists in both an affiliate registry and the UDDI Business Registry. This could have occurred in several different ways. A publisher in Affiliate A may have subscribed to a certain set of entities in the root registry, publishing them in itself on an ongoing basis as changes are received. Or, a publisher in Affiliate A may have generated a set of entities in its local registry and then published them to the UBR. Such data exchange is potentially a two-way process: the data may originate in the root and end up in Affiliate A, or it may originate in Affiliate A and end up in the root.

In Scenario (3), the two affiliates share data. Again, this sharing has several permutations. Publishers in Affiliate A and B may have an explicit agreement to subscribe to one another's data, or a publisher of one of the registries may have simply copied a set of data from another registry, unbeknownst to that registry, and published the data to its registry.

Scenario (4) brings the interaction among these three registries to a zenith: there is the case where all three registries share a common set of data, such as key generator tModels. Again, the circumstances and mechanisms through which such a common dataset is established are manifold.

Given the above exercise, it is critical both for nodes of registries and for clients of registries to understand the different ways in which inter-registry communication can occur. In particular, understanding how the establishment of key generators within different registries regulates the process of inter-registry publication is paramount.

8.2 Data Management Policies and Procedures Across Registries

8.2.1 Establishing a Relationship with a Root Registry

If a registry intends to allow the users of that registry to copy entities into other registries, some considerations need to be taken when that registry goes online, before it generates a single key. This is crucial because if the registry begins to generate keys before establishing a key generator with a root registry, the registry may not be able to become an affiliate later on.

To bring up each node in a registry affiliated with a root registry, the node should begin by publishing a new key generator tModel of type keyGenerator[33] in the root UDDI registry. The node should then save this tModel in itself exactly as it has been saved in the root registry. This tModel then becomes the key generator tModel for the new node. Owning the imported tModel gives the new node the capability it needs to generate new keys that are guaranteed to be unique within the root registry by deriving them from its key generator.

During normal operation, nodes need to generate keys – i.e., while saving new entities for which keys have not been proposed by their publisher. They do this by deriving them from key generators they own. Nodes may choose any algorithm they wish to generate the key specific string (KSS)[34] that makes each of the derived keys unique. Each node is responsible for managing the uniqueness of the keys it generates for itself. One way to do this is to use UUIDs for the extensions it generates. Another is to use sequential integers.

Once a key generator is established with the root registry, the affiliate registry can begin safely generating keys that are unique in the context of the root registry and unique in any registries

[33] See section 5.3.18.3.1 Domain Key Generator tModels

[34] See section 5.3.2 Publishing Entities With Publisher Assigned Keys

that share an affiliation with the root registry and have followed the recommended keying policy.

For example, imagine Registry X is established by a company as an affiliate of the UDDI Business Registry. The following steps would occur:

1. To begin Registry X would publish a tModel of type keyGenerator in one of the UBR nodes. The tModel might be as follows: uddi:AC104DCC-D623-452F-88A7-F8ACD94D9B2B:keygenerator. Note that this example uses a uuidKey for its key partition. It could have used a domainKey as well.
2. Then, Registry X would save that tModel into its own registry.
3. Additionally, Registry X would establish that all entities saved to Registry X that need a new key generated would be programmatically derived from uddi:AC104DCC-D623-452F-88A7-F8ACD94D9B2B:keygenerator. For example, Registry X might use sequential numbering to perform this task, so that the first businessEntity saved into node x would get key uddi:AC104DCC-D623-452F-88A7-F8ACD94D9B2B:1, the 2nd would get the key uddi:AC104DCC-D623-452F-88A7-F8ACD94D9B2B:2 and so on.

A consequence of the way key generation works is that if a registry exports a key generator tModel from some other registry and assigns it to a publisher, that publisher may use that tModel to generate keys in that registry. In other words, the procedure outlined above can be used between affiliate registries as well.

Note: If a registry has started generating keys without first becoming an affiliate of a root registry and its publishers still wish to share data with another registry, the registry would need to become an affiliate and then re-key its data with keys derived from the registry's key generator. In the case where the registry has begun generating keys as a v1 or v2 registry, the keys will need special consideration[35].

8.2.2 Data Sharing

The actual operation of sharing data between registries is accomplished using the existing API sets. The Inquiry API and/or the Subscription API are used to get data from a registry; the Publish API is used to promote entities into a registry. A new set of terms is introduced to help clarify the behavior:

A *source* registry is a registry with data to share.

A *target* registry is a registry that will consume data.

An *importer is a publisher who* reads data via the Inquiry or Subscription API from one or more source registries and publishes it to a target registry. Likely, importers are software applications that use the UDDI APIs to achieve this multi-step process.

8.2.2.1 Sharing Data

Importers copy entities from one or more source registries and publish them, potentially with modification[36], in a target registry. Given its description, every importer is a publisher in its target registry and an inquirer in its source registries. Importers can read their source registries by subscribing to them or by simply issuing find_xx and get_xx Inquiry APIs calls.

For an importer to do its work it must own the requisite key generator tModels in the target registry, and the target registry must have a policy that allows publisher-assigned keys. In this

[35] See Chapter 10 Multi-Version Support

[36] Signed entities may not be modified without breaking the signature

way, the importer can preserve the keys of the entities it reads from the source registry and publishes to the target registry.

An important case for which this is particularly simple is the one in which the importer is a part of the operation of an affiliate, the source registry is that affiliate, the target is the root, and the data to be imported has keys based on the affiliate's key generator. Since the affiliate already owns the requisite key generator in the target, and the root, by definition, accepts publisher-assigned keys, the required conditions are met. This scenario is visually represented by case 2 in the diagram above.

Another important but simple case is one in which the source registry is the root, the target is an affiliate, and the importer is a part of the operation of the target registry. Since the target is the importer, it is granted the authority to publish root key generators and entities with uuid keys. With these permissions, the importer can import any entities from the root that the target's policy dictates. Because the importer is still bound by the rules for key generation – in particular because it is not permitted to generate keys in a partition for which it does not own the key generator – the importer cannot cause key conflict. This scenario is also visually represented by case 2 in the diagram above; but with the reverse data flow.

In the case where both the source and the target are affiliates, the process may not be as simple. First, the target registry must have a policy that allows it to accept publisher-assigned keys. Assuming this is the case, the importer must own the necessary key generator tModels in the target registry. The recommended way for the importer to obtain the required ownership is to have the target registry import key generators from the root registry and then transfer ownership of them to the importer. In other words, out-of-band communication is required that cannot be conducted simply by a save_tModel call from the importer. The out-of-band communication is important because it provides the target registry with an opportunity to verify that the importer should be allowed to be the proxy in the target for whoever owns the key generator in the root.

Consider an example. An importer wants to copy data from Registry K to Registry J, both of which are affiliates of the UDDI Business Registry. The importer must get Registry J to grant it the authority to be a publisher and must get Registry J to give it ownership of the key generator(s) it needs to publish the entities it wishes to import from Registry K. If Registry J trusts the importer, it grants the importer publishing authority, imports the key generator tModels requested by the importer from the root, and transfers ownership of them from itself to the importer. Typically, the key generators the importer needs include the key generator(s) used by Registry K when it generates keys.

Phrased differently, target registries should establish a trust relationship with those it grants ownership of the key generators it imports. By granting a publisher ownership of an imported key generator, the target registry is granting the publisher the authority to be the representative in the target registry of whoever owns the key generator in the root.

9 Policy

Policies within UDDI are statements of required and expected behavior. Within the UDDI architecture there are registries which are composed of one or more nodes. The registry defines the domain of the policy for the nodes that make it up and may delegate the definition of a particular policy to one or more of the nodes within its domain. Within policy then, there is a hierarchical relationship between registry policies and node policies including whether a registry allows nodes to specify polices. Registries MUST also identify the Policy Decision Points. Policy Enforcement Points are responsible for enforcing the policies and these MAY be the same as the Policy Decision Point.

Registries may also affiliate. Affiliated registries are sets of registries that share compatible policies for assigning keys and managing data. (See Section 1.5.5.*Affiliations of Registries*)

9.1 Definitions

See Appendix L *Glossary of Terms* for definitions of terms appearing in **bold**. Many of these are derived from general terms in RFC 3198, the IETF glossary for policy. See http://ietf.org/rfc/rfc3198.txt.

9.2 Policy

This specification defines **policy abstractions**. Registries MUST define a policy rule for each of the policy abstractions. Each **policy rule** is decided by a **Policy Decision Point** (PDP). For some policy abstractions the PDP MUST be the registry. Part of each policy rule is to designate the **Policy Enforcement Points** (PEP) that is responsible for enforcing the policy. The PEP MAY be the same component as the Policy Decision Point.

The combination of policies and implementation for a UDDI registry should form an explicit security model that is available for evaluation in a risk assessment. It is recommended that implementers of UDDI nodes and registries and users of those registries obtain or perform a risk assessment of the implementation.

In the following Section 9.4 *UDDI Registry Policy Abstractions*, the registry policy abstractions are defined in a narrative form. Other policies MAY be delegated to the node and these are defined in Section 9.5 *UDDI Node Policy Abstractions*. At the end of the document is a summary table which captures the abstractions in a way that cross references to the narrative text and includes some additional information.

A registry MAY specify additional policy abstractions and rules. A registry MAY also allow nodes in the registry to specify additional policy abstractions and rules. Representation of the policy rules corresponding to policy abstractions in this chapter and any additional policy abstractions SHOULD be communicated as described in the following section.

9.3 Representation of Policy

There are two types of polices defined in the following sections. There are policies that can be communicated through representation in an XML document and there are policies that can be modeled within UDDI through the use of UDDI elements. Section 9.7 *UDDI Policy Summary* recommends a method (document or model) for conveying each policy.

If a policy abstraction can be modeled, Section 9.4 and Section 9.5 describe how this MAY be done. Policies that impact the configuration of UDDI clients SHOULD be modeled in the UDDI data structures.

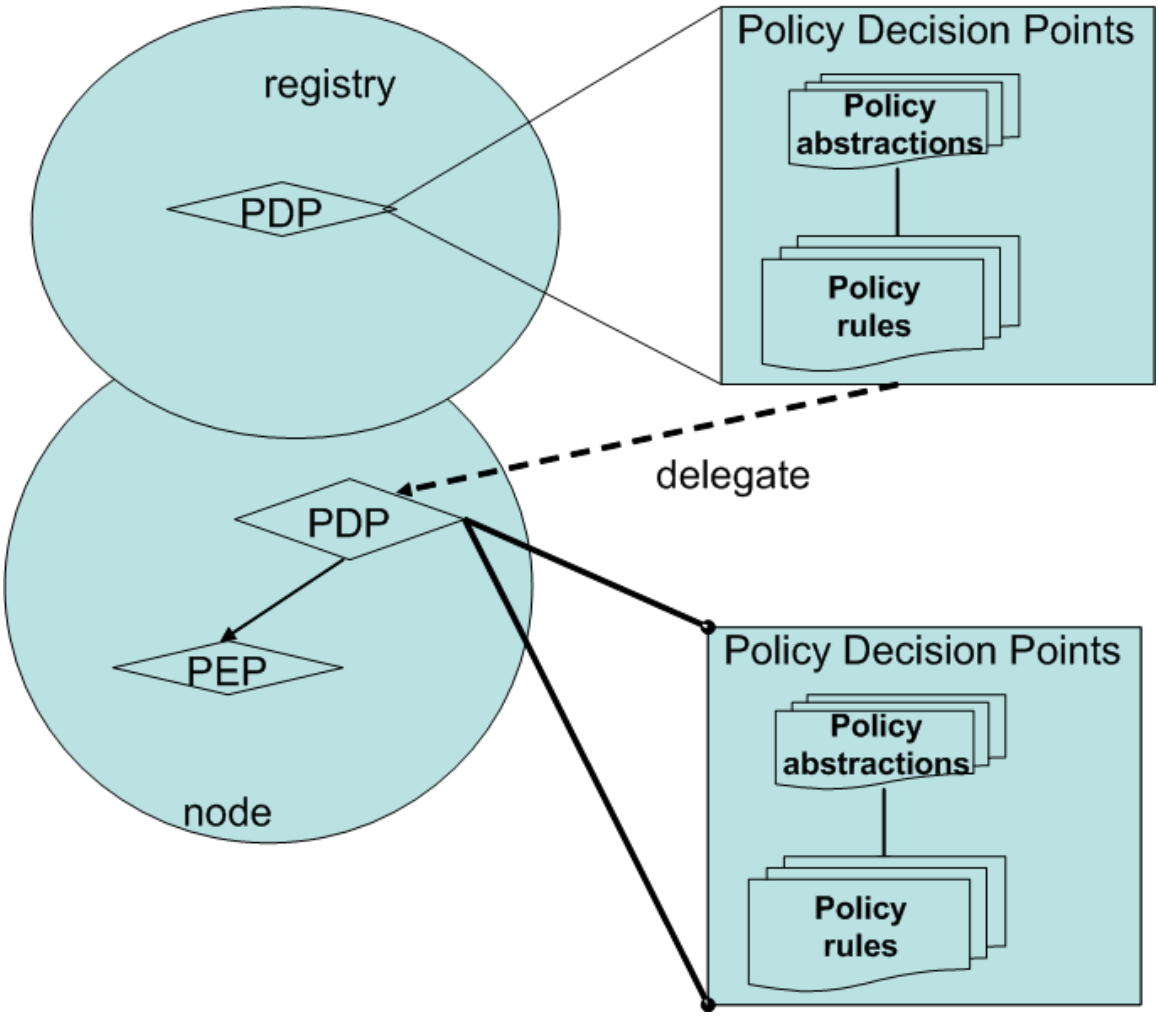
Policies communicated through an XML document SHOULD include the name of the policy abstraction from this section, a description of the policy rule and a declaration of the policy decision point and the policy enforcement point. An XML Schema for creating this document has been created in the namespace urn:uddi-org:policy_v3 and is described below in Section 9.3.1 *Policy Schema.*

When documenting policies, there is a hierarchy in the way policies are defined. A registry defines the broad registry policy abstractions, one of which is whether or not a policy may be defined by the individual nodes within the registry. If a registry allows nodes to specify policies it is said to be "delegating" the policy expression to the node.

If policies are "delegated" to nodes to specify, the node may specify the abstraction or it may specify policy rules.

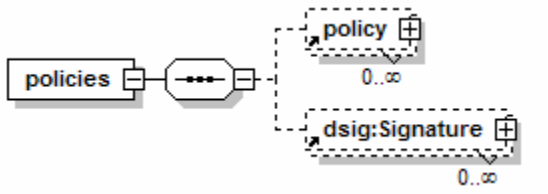
The picture below illustrates the relationship between a registry and its nodes. The registry as the Policy Decision Point for the registry and its nodes defines its policy abstractions and rules.

If the registry allows delegation, the nodes may also define policy abstractions and rules, but these must be consistent with the policies inherited from the registry.

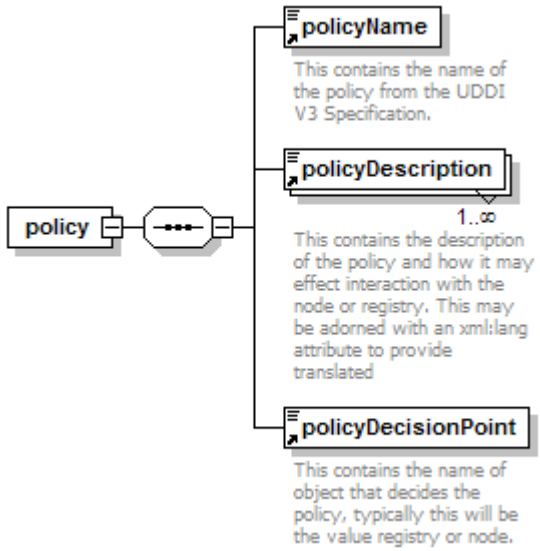


9.3.1 Policy Schema

The policies element illustrated below represents all of the policies as a sequence of policy elements. Each policy element in the sequence represents one policy listed in the policy abstractions that are applicable to the UDDI API set. In the policy document instance referenced by the policy service, there should be one instance of a policy element for each policy that applies to all API sets provided by the node. The policies element MAY also contain an XML Signature using the dsig:Signature element. Signing a policy document is described in Appendix I.



A policy element represents one individual policy. It contains the name of the policy abstraction in the policyName element, a string describing where the policy is decided in the policyDecisionPoint element and the actual description of the policy in one or more languages in the sequence of policyDescription elements.



The policyName element contains the policy name defined in the policy abstractions below in Section 9.4 and 9.5.

The policyDescription element contains a description of the effect of the policy implementation. This element can be adorned with the xml:lang attribute and can appear multiple times to allow for translations of the policy description.

The policyDecisionPoint element contains a string describing where the policy is decided. Typically, this value will be either the string "node" or "registry". Use of an alternative policy decision point should be described in this element.

9.3.2 Policy Documents

An instance of the policy document should be a Web accessible document and the URL for retrieving this document should be included in the overviewDoc element in the instanceDetails element of a tModelInstanceInfo element referencing a UDDI API set tModel. The contents of the elements in the policy document are intended to be human readable.

Appendix K contains an example of an instance of a policy document and the corresponding Node Business Entity element for the implementation of a UDDI API set. The instance of the policy document that is associated with each API set SHOULD include all policies of the registry and of the node that are applicable to the API set.

It is RECOMMENDED that the instance of the policy XML Document(s) conform to the XML Schema for policy in the namespace urn:uddi-org:policy_v3. A policy document SHOULD be provided for each UDDI API set binding and the location of the policy document SHOULD be an HTTP GET accessible document with the URL identified in the overviewURL element of the overviewDoc in the instanceDetails element of a tModelInstanceInfo element referencing a UDDI API set tModel.

In the event that another form of policy documentation format or standard for policy declaration captures a policy decision or decisions in a manner more appropriate to the particular node and registry, it is recommended that this standard be registered as a tModel with the overviewURL identifying the documentation for the document format. Registering a tModel allows different policy document formats and standards to be used by a particular implementation of a UDDI node or registry. The actual instance of the policy document SHOULD be provided for each UDDI API set binding where the policies will be enforced. In the bindingTemplate element, the location of the policy document SHOULD be an HTTP GET accessible document with the URL identified in the overviewURL element.

9.3.3 Policy Service within UDDI

By using the UDDI model for defining Web services it is possible to define a policy service which supplies the documented policies of the registry and node.

It is RECOMMENDED that a UDDI node provide a separate businessService of the Node Business Entity that represents the general policy service. This businessService MUST have one bindingTemplate that references the uddi-org:v3_policy tModel described in Section 11.5.7 *UDDI Policy Description Specification*. This policy service is described below in Section 9.3.3 *Policy Service within UDDI*.

9.3.4 Policy Modeling

Policy decisions that can impact the configuration of a UDDI client SHOULD be represented in UDDI data structures for discovery by UDDI clients. The modeling of policy makes use of the data structures of UDDI by representing policy concepts as tModels. Use of the concept by an implementation of a UDDI API set is reflected by including the tModel in the bindingTemplate for the UDDI API set. Options within a concept are represented as instance parameters in the instanceParms element. Instance parameter values that are related to options described in the UDDI Version 3.0 Specification are included in the UDDI Version 3.0 Policy Instance Parameters schema which is in the namespace, urn:uddi-org:policy_v3_instanceParms

Support of an API set, described in Chapter 5, by a node as part of this procedure MUST be represented in a bindingTemplate in each Node Business Entity that supports it.

As multiple policy instance parameter elements need to be reflected in the instanceParms element, the container element, UDDIInstanceParmsContainer, MUST be used to wrap other elements as defined in the Policy Instance Parameters schema.

It is important to note that XML in instanceParms MUST be encoded as described in Section 3.5.2.4 *instanceDetails.*

See Appendix K *Modeling UDDI within UDDI - A Sample* for examples.

9.4 UDDI Registry Policy Abstractions

This section includes the highest level of policy abstractions, the registry policy abstractions. Since registries may be composed of nodes, the next level of abstraction is the node

abstraction which is described in Section 9.5 *Node Policy Abstractions*. In some cases there are UDDI policies that are "recommended". These are documented in Section 9.5.16 *UDDI Recommended Registry Policies*.

9.4.1 Registry Policy Delegation

A registry MUST specify whether or not it supports delegation of policies to the nodes within it. The definition of policies that a registry identifies MAY be delegated to a node to specify a node-specific policy. This is indicated in the summary table as "may be delegated".

9.4.2 Registry General Keying Policy

UDDI Version 3 introduces the ability for publishers to generate entity keys. This feature preserves the requirement for distinct keys while enabling keys to be created in a safe, efficient manner.

A registry MUST have a policy on key format and key generation. The registry MUST have a policy on data integrity or how the keyspace is protected from unauthorized modification.

Registries MAY use whatever keying policies they wish subject only to the constraints that keys MUST be URIs, and the registry MUST have policies that prevent key collisions.

9.4.3 UDDI keying scheme

This specification presents a keying scheme (in Section 4.4 *About uddiKeys*) that all registries MUST use. Policy decisions MUST be defined for the following policy abstractions.

9.4.3.1 Registry Key Generation for Nodes

Registries MUST establish policies for deciding whether a given node is permitted to publish a given key generator tModel. Section 4.4 *About uddiKeys* contains details on key generation. See Section 5.2 *Publication API Set* for details on publish APIs.

9.4.3.2 Registry Key Generation for Publishers

Registries MUST establish policies for deciding whether publishers are permitted to publish key generator tModels. Section 4.4 *About uddiKeys* contains details on key generation. See Section 5.2 *Publication API Set* for details on publish APIs.

Also, registries MUST establish policies for the criteria by which a given publisher is allowed to register a given key generator tModel. Chapter 5 *UDDI Programmers APIs* also contains details on key generator tModels and this section should be reviewed before creating the key generation policy.

9.4.3.3 Registry Key Default

The registry MUST specify what the policy is when a key is not supplied on an API, but MAY delegate the generation of default keying to its nodes.

9.4.3.4 Registry Support of uuidKeys

Another policy decision is whether nodes will accept a key generator tModel that is NOT a domainKey, but is a uuidKey.

9.4.3.5 Root Key Generation

In order for an affiliation to be established among UDDI registries, the registries involved MUST have compatible policies regarding key generation. The keying scheme facilitates this by designating one of registries in the affiliation as "the root registry". The root UDDI registry serves as the naming authority by assigning first-order or "root" partitions of the space of keys

to registries and others who need to generate keys. All registries that participate in that affiliation look to the root registry as the source on which to establish the uniqueness of a given set of keys. An example of a root registry is the UDDI Business Registry. See Section below on UBR policies to see the details of the UBR Root Key Generation Policy.

9.4.4 UDDI Information Access Control Policy

The goal of a UDDI registry is to be a useful, reliable registry of business services. To achieve this purpose each registry must establish the policies needed to appropriately protect the information in its possession. The policies that a registry defines to protect business information is its information management policy.

UDDI nodes that maintain custody of UDDI information and implement a data storage mechanism are responsible for the Data Model of the underlying storage of the data elements and its mapping to the Information Model.

The mapping to the information model is represented by the implementation of the API sets defined in this specification. This information model MAY be extended as described in Appendix H *Extensibility*. When an extension to the information model is used, it MUST be represented by a different tModel as described in Appendix H and the extension MAY have a significant impact on interpretation of the information and the policies of the registry. Use of an extension MAY be prohibited by a registry, and if allowed, it is STRONGLY RECOMMENDED that an impact of an extension on the information model of UDDI is assessed by both operators and users of the registry.

9.4.5 Adding nodes to a registry

A registry MUST specify how nodes are added to and deleted from the registry. If a registry supports the removal of nodes, then it MUST specify how the node custody will be transferred and how the data migration will occur.

If a registry uses configuration information for tracking the nodes of the registry, then it must state how this configuration information is protected from unauthorized access. Protection of information is provided through access control, data integrity and data confidentiality policies.

9.4.6 Person, Publisher and Owner

When publishing information in a UDDI registry the information becomes part of the published content of the registry. During publication of an item of UDDI information, a relationship is established between the publisher, the item published and the node at which the publish operation takes place. A registry may be composed of more than one node. The node to which the information was submitted and accepted via the UDDI v3 publication API set is the custodian of the item. A registry MUST specify whether transfer of ownership is supported and how this is accomplished.

9.4.6.1 Registry Registration Policy

A registry MUST specify how individuals who want to publish information become known to the registry. The registry may delegate registration to its nodes.

9.4.6.2 Publication Limits

When defining who can publish information, a registry may also constrain the amount of information that an individual can publish and through what means. A registry may establish different limits for different classes of users. These are known as "tier limits". This policy may be delegated to the nodes in a registry.

9.4.7 Transfer of Ownership

The registry must define whether intra-node ownership transfer is a service that is supported. If it is supported, the node must ensure that the UDDI information model is preserved while the data model is modified. A process for ownership transfer that MAY be supported by a registry or node is outlined in Section 5.4 *Custody and Ownership Transfer API Set*.

9.4.7.1 Modeling of Ownership Transfer Support

It is RECOMMENDED that the Custody and Ownership Transfer API Set described in Section 5.4 be used for intra-node ownership transfer of businesses and tModels. If this recommended API is supported by a node, this MUST be represented as an instance of the uddi-org:custody_v3 tModel in a bindingTemplate in the Node Business Entity. Use of a different API set for ownership transfer is outside the scope of this specification, but should be modeled as a tModel referenced by a bindingTemplate in the Node Business Entity.

9.4.8 Registry Authorization Policy

A registry MUST have a policy on access to the information registered in it. A registry may specify a policy of global access for all API's or it may specify a different type of access for each API and/or each publisher. The registry policy MAY specify that the authInfo element is required for an API. The policy describing who has access to what information is called the authorization policy. The implementation of each API, as reflected by the bindingTemplate that describes it (see section below), is a Policy Enforcement Point.

9.4.8.1 Delegation of Authorization Policy

A registry MAY allow nodes to specify their own access policies (delegation of policy), but an individual node's access policy MUST be consistent with that of the other nodes in the registry and MUST NOT compromise the data in the registry as a whole.

If the registry policy for authorization requires a unique identity for each owner of UDDI data, the registry MAY delegate the registration to each node in the registry but the registry MUST specify how the registration maps to the authorization policy.

9.4.9 Modeling Authorization

A policy that requires authorization to use a particular UDDI API set SHOULD be represented in the structures of the Node Business Entity element. An authorization concept used by an implementation of a UDDI API set SHOULD be represented by a tModel. This tModel SHOULD be referenced in the bindingTemplate implementing a UDDI API set that uses the particular authorization concept. An example of this modeling would be a tModel representing an authorization concept that requires the use of authentication credentials transmitted as HTTP headers. An implementation using this concept in conjunction with the publication API would be represented by a bindingTemplate containing both a tModelInstanceInfo for the publication API set tModel and a tModelInstanceInfo for the tModel for the authorization concept.

In addition to authorization methods that are outside the scope of this specification, several of the UDDI API sets allow implementations to enforce authorization using an authInfo element obtained through the Security API set described in Section 5.3 *Security Policy API Set*. The bindingTemplate element representing each UDDI API set implementation SHOULD indicate if the authInfo element is required, optional or ignored in the bindingTemplate for the API set. It is RECOMMENDED that an XML document be inserted in the instanceParms element of the bindingTemplate element that represents an implementation of the Inquiry, Publication or Subscription API set. The XML element that SHOULD be in the instanceParms XML document is an instance of one of the NMTOKEN values for authInfoUse included in the UDDI v3 Policy Instance Parameters schema:

```
<authInfoUse xmlns="urn:uddi-org:api_v3_instanceParms">required</authInfoUse>
<authInfoUse xmlns="urn:uddi-org:api_v3_instanceParms">optional</authInfoUse>
<authInfoUse xmlns="urn:uddi-org:api_v3_instanceParms">ignored</authInfoUse>
```

9.4.10 Registry Data Integrity

A registry MUST specify how it maintains the information registered in it. The nodes MUST enforce this policy.

9.4.11 Registry Approved Certificate Authorities

Each registry MUST establish the certificate authorities it recognizes. A registry MAY delegate this policy to a node.

9.4.12 Registry Data Confidentiality

A registry MAY have a policy for protecting the information under the custody of its nodes from unauthorized access. This policy has two dimensions.

9.4.12.1 Persistent Data Confidentiality

A registry MAY specify a policy for the encryption of UDDI data when stored .This policy MAY be delegated to the node to implement and is usually referred to as persistent data confidentiality

9.4.12.2 Transient Data Confidentiality

The data supplied in an API MAY need to be protected from being "sniffed" on the wire while being transmitted. This confidentiality (or the encryption of the information) MAY be specified as part of the transport of the API. Each API set MAY have different policies for Data Confidentiality.

It is also possible to extend the transport of the information model in UDDI. This allows other transport protocols and extensions of SOAP to be used by an implementation. The impact of such an extension may, and will likely impact the information model presented in this specification. These extensions MUST be represented in the UDDI structures as tModels, or concepts, which include documentation to explain the concept of the extension or substitution of a transport protocol. Extension or substitution of any portion of the normative or recommended mechanisms in this specification is treated as an extension or substitution of the UDDI information model. In accordance with extensions to the information model, it is STRONGLY RECOMMENDED that an impact of an extension on the information model of UDDI is assessed by both operators and users of the registry.

9.4.12.3 Modeling Data Confidentiality

The RECOMMENDED means of conveying the policy for data confidentiality in transmission is to include a tModelInstanceInfo referencing a tModel that represents the mechanism for confidentiality in the binding for the UDDI API set. One example of modeling data confidentiality in transmission is represented by the application protocol tModel in Section 11.3.1 *Secure Sockets Layer Version 3 with Server Authentication*.

9.4.13 Registry Audit Policy

A registry MAY specify a policy for the recording of information to maintain an account of the activity that has been processed. The audit policy MAY be delegated to the nodes in a registry. The audit policy SHOULD state what actions are audited and under what conditions. Also, the audit policy SHOULD state who has access to the audit trail. Audit policies could conceivably

need to be presented as evidence in a legal proceeding and a UDDI registry SHOULD have a risk analysis done in order to assess the critical information that needs to be recorded.

9.4.14 Registry Privacy Policy

A registry MAY specify a policy for protecting the information collected about users of the registry. The privacy policy MAY be delegated to the nodes in a registry.

9.4.15 Registry Clock Synchronization Policy

The degree to which the clocks of each UDDI node are synchronized is a matter of registry policy. The clock is used to generate the created, modified and modifiedIncludingChildren elements. The frequency with which each clock is incremented (e.g.: 60 Hz, 100 Hz, etc) is also a matter of registry policy.

9.4.16 Registry Replication Policy

When a registry consists of a single node, replication is not required. If the registry consists of multiple nodes, then a policy for replication of the information in each node to every other node of the registry MUST be specified.

9.4.16.1 Registry with Single-Master data model

Registries that use the replication protocol defined in Chapter 7 *Inter-Node Operation* to replicate data use a single-master data model. See the section below on UDDI Node Policy Abstractions for recommendations for a single-master data model of replication.

9.4.17 Support for Custody Transfer

A multi-node registry MUST establish a policy stating if it allows transfer of custody of its data from one node in the registry to another node.

9.4.17.1 Modeling Custody Transfer

It is RECOMMENDED that the Custody Transfer API Set described in Section 5.4 be used to initiate inter-node custody transfer. If this recommended API is supported by a registry this MUST be represented as instances of the uddi-org:custody_v3 tModel in bindingTemplate elements in each participating node's Node Business Entity.

It is further RECOMMENDED that the transfer_entities API described in Section 5.4 be used to communicate the custody transfer between nodes. If this recommended API is supported by a registry this MUST be represented as instances of the uddi-org:custody_ transfer_v3 tModel in bindingTemplate elements in each participating node's Node Business Entity

Use of a different API set for custody transfer is outside the scope of this specification but should be modeled as a tModel referenced by bindingTemplate in the node's Node Business Entity.

9.4.18 Registry Subscription Policy

A registry must define a policy for supporting subscriptions including whether nodes may define their own policy. Individual nodes, including those in the UDDI Business Registry MAY establish policies concerning the use of the subscription API. A registry that supports subscription MUST contain at least one node that has a bindingTemplate referencing the uddi-org:subscription_v3 tModel in its Node Business Entity.

9.4.18.1 Registry Limits on Volume, Renewal and Retries

Such policies might include restricting the use of subscription, defining which APIs are supported, establishing authentication requirements for subscriptions, or even imposing fees for the use of subscription services.

9.4.18.2 Subscription Duration

The duration or life of a subscription is also a matter of node policy. Subscribers may also create multiple subscriptions and registries may impose limits on the number or type of subscriptions subscribers may create.

9.4.18.3 Authorization for Subscription

A registry MUST specify a policy for deciding who is able to create, subscribe to and receive subscriptions including whether each node may have its own policies on subscription. The policies that include authorization SHOULD be reconciled with other authorization policies including the registries policy for authorization of APIs.

9.4.18.4 Modeling Subscription

A registry that supports subscription MUST contain at least one node that has a bindingTemplate referencing the uddi-org:subscription_v3 tModel in its Node Business Entity.

The bindingTemplate element describing the subscription API set implementation SHOULD indicate whether find API elements as a filter are supported by the implementation. It is RECOMMENDED that an XML document be inserted in the instanceParms element of the bindingTemplate that represents an implementation of the Subscription API set. The XML element that SHOULD be in the instanceParms XML document is an instance of one of the NMTOKEN values for filterUsingFindAPI included in the UDDI v3 Policy schema:

```
<filterUsingFindAPI xmlns="urn:uddi-org:policy_v3_instanceParms>
   supported</filterUsingFindAPI >
<filterUsingFindAPI xmlns="urn:uddi-org:policy_v3_instanceParms>
   unsupported</filterUsingFindAPI>
```

It is important to note that XML in instanceParms MUST be encoded as described in Section 3.5.2.4 *instanceDetails.*

9.4.19 Registry Value Set Policies

UDDI allows for the creation of category, identifier, and category group systems and allows this information to be referenced within the registry. The node is the enforcement point for value set policies. The node MUST respond with an E_unsupported error code to requests to publish which include references to the unsupported checked value set tModels.

9.4.19.1 Value set delegation policy

The registry MUST have a policy on whether or not it allows individual nodes to specify their own policies for value sets.

9.4.19.2 Checked value sets policy

Value sets can be checked or unchecked. A registry must decide if it supports checked value sets. If checked value sets are allowed, the registry MUST have a policy for differentiating checked and unchecked value sets. Further, the registry MAY have a policy for determining which checked value sets it supports.

9.4.19.3 Uncached checked value sets policy

If checked value sets are allowed, the registry MUST have a policy for determining whether uncached checked value sets are supported.

9.4.19.4 Cache invalidation policy

If cached checked value sets are supported, the registry must establish a policy for detecting a need for cache invalidation.

9.4.19.5 External validation Web service supported policy

If uncached checked value sets are supported, the registry must establish a policy for determining whether external validation Web services are supported.

9.4.19.6 Value set Web service discovery policy

If checked value sets are supported, the registry must establish a policy for modeling external value set caching and/or validation Web services, and their means of discovery.

9.5 UDDI Node Policy Abstractions

This section describes the policy abstractions that a registry MAY delegate to a node to define. These node policies need to be consistent with those of the registry.

9.5.1 Node Key Generation

If a registry delegates Key Generation, the nodes MUST establish policies for deciding whether publishers MAY publish a given key generator tModel. Section 5.2.2 *Publishing entities with publisher-assigned keys* contains details on key generation. See section 5.2 *Publication API Set* for details on publication APIs.

If delegated, the node MUST specify what the policy is when a key is not supplied on a publication API.

9.5.2 Node Publisher Generated Key Assertion

Each node must establish whether it will accept publisher generated keys. Nodes may accept a key generator tModel that is NOT a domainKey but is a uuidKey.

9.5.3 Node Information Policy

UDDI nodes that maintain custody of UDDI information and implement a data storage mechanism are responsible for the Data Model of the underlying storage of the data elements and its mapping to the Information Model.

9.5.4 Node Authorization Policy

If a registry allows nodes to specify their own access policies (delegation of policy), an individual node access policy MUST be consistent with the other nodes in the registry and MUST not compromise the data in the registry as a whole.

If the registry policy for authorization requires a unique identity for each owner and is delegated to the node, each node in the registry MUST specify how the registration maps to the authorization policy.

9.5.5 Node Registration and Authentication

The node, as custodian of registry information MUST have a policy on what publishers are known to it. This is called the registration policy. The registration policy MUST support the

implementation of the authorization policy. The registration policy MAY specify that all access to the information is public. The registration policy MAY specify that all users are required to authenticate to the node before API access is allowed.

The node MUST specify whether (or under what conditions) any meta information about a publisher is accessible (name, company, phone number, etc). The protection and release of such meta information MAY also be considered to be a privacy policy.

9.5.6 Node Publication Limits

A node MAY chose to establish limits for the amount of information publishers are allowed to publish.

9.5.7 Node Policy for Contesting Entries

Each node SHOULD establish a convention for contesting entries.

9.5.8 Node Audit Policy

The audit policy MAY be delegated to the nodes in a registry.

9.5.8.1 Node Availability Policy

Each node SHOULD make available its policies for UDDI service availability. Each node SHOULD make clearly visible planned outage and maintenance schedules.

9.5.9 Node Collation Sequence Policy

Each node MUST specify the default collation sequence which it supports. A node MAY specify support for optional additional collation sequences. All collation sequences are specified via use of sortOrder tModels.

9.5.9.1 Modeling Sort Orders

The tModels for all supported sort orders SHOULD be included in the bindingTemplate for the UDDI Inquiry API set. The default sort order SHOULD be indicated as an instance parameter using the defaultSortOder element declared in the UDDI v3 Policy Instance Parameters schema

```
<defaultSortOrder xmlns="urn:uddi-org:api_v3_instanceParms>
   binarySort</defaultSortOrder>
```

9.5.10 Find Qualifier Policy

Each node MUST specify the find qualifiers which it supports on the Inquiry API Set and Subscription API Set. A node MAY specify support for optional additional find qualifiers not documented in this specification by registering them as a tModel. All find qualifiers are specified via use of findQualifier tModels.

9.5.10.1 Modeling Find Qualifiers

The tModels for all supported find qualifiers beyond those listed as required by UDDI implementations in Section 5.1.4 SHOULD be included in the bindingTemplate for the UDDI Inquiry API and Subscription API sets. All find qualifiers in Section 5.1.4, with the exception of the OPTIONAL diacriticInsensitveMatch find qualifier, are supported by a UDDI Web service referenced in a bindingTemplate which references the Inquiry API set tModel or a Subscription API set tModel where a find API is supported as a filter.

9.5.11 Node Approved Certificate Authorities

Each node MUST establish the certificate authorities it recognizes.

9.5.12 Node Subscription API Assertion

The Subscription API is an optional API that may or may not be implemented by a node. Each node MUST determine whether subscription is in fact supported and, if so, the exact kinds of subscription that are supported.

9.5.13 Node Element Limits

A node may limit the number of <name>, <personName> or <description> elements decorated with the same xml:lang attribute. So, for example, a node may reject a message if the node policy is to only allow two name elements with the same language and more than two elements are supplied.

An additional limit that a node may impose is a limit on the size of requests to a particular API. The limit should be expressed as the maximum number of bytes that will be accepted for subsequent processing in one request message.

9.5.13.1 Modeling Request Size Limit

The bindingTemplate element describing the API which limits the size of a request message SHOULD indicate the limit in number of bytes in the instanceParms element. The XML element that SHOULD be in the instanceParms XML document is an instance of one of the maximumRequestMessageSize elements included in the UDDI v3 Policy schema:

```
<maximumRequestMessageSize xmlns="urn:uddi-org:policy_v3_instanceParms>
   2000000</maximumRequestMessageSize>
```

It is important to note that XML in instanceParms MUST be encoded as described in Section 3.5.2.4 *instanceDetails*.

9.5.14 Node HTTP GET Services

A node MAY specify if it supports an HTTP GET service for access to the XML representations of UDDI data structures. If the node does offer such a service, it SHOULD specify the base URI to be used. The base URI SHOULD be specified in the policyDescription element of the policy element for the "HTTP GET Support" policy. See Section 6.5 *Node HTTP GET Services*.

9.5.15 Node discoveryURL Generation

A node MAY establish a policy on whether it generates and adds discoveryURLs to businessEntity elements. This is NOT RECOMMENDED behavior as it complicates the use of digital signatures.

9.5.16 Node XML Encoding Policy

A node MAY establish a policy on whether it uses UTF-8 or UTF-16 as the character encoding it uses in response messages so that client-side XML document processing can be optimized.

9.6 UDDI Recommended Registry Policies

The policies listed in this section are those that are RECOMMENDED in order to provide an example of a UDDI Registry instance. Each registry MAY chose to define its own policies but should be cautious and understand the relationships between policies, registries and nodes.

.

9.6.1 Key Generator tModels

It is RECOMMENDED that the saving of a key generator tModel is disallowed and rejected from v1 and v2 clients.

9.6.2 Information Model

The UDDI information model is instantiated in the data model through the descriptions of the data structures (see Chapter 3 *Data Structures*), operations (see Chapter 5 *UDDI Programmers APIs*) and the policies described in this section.

A registry MAY offer different bindings for the APIs across the constituent nodes of the registry. A particular node in a UDDI registry MAY provide access to the UDDI data through one or more UDDI API sets. A UDDI node MAY also support any number of bindings for the number of APIs it offers. Variations of policies for these different bindings MAY be different. For example, a node might choose to offer both http and https bindings for the inquiry API and apply different access policies depending on which binding is used.

9.6.2.1 Modeling UDDI APIs

A UDDI registry MUST have at least one node that offers a Web service compliant with the Inquiry API set. A UDDI registry SHOULD have at least one node that offers a Web service compliant with the Publication, Security, and Custody and Ownership Transfer API sets. If a UDDI registry has multiple nodes, all nodes SHOULD offer Web services that are compliant with the Replication API set. The Subscription and Value Set API sets are OPTIONAL for all nodes and all registries.

The availability of one or more UDDI API sets at a node SHOULD be reflected through a bindingTemplate referencing the appropriate UDDI API set tModel for each Web service endpoint.

API Set	tModel	Recommended Transport	Recommended Security Mechanisms Integrity / Confidentiality	Authentication
Inquiry	uddi-org:inquiry_v3	HTTP		
Publication	uddi-org:publication_v3	HTTP	SSL V3	
Security	uddi-org:security_v3	HTTP	SSL V3	
Custody Transfer	uddi-org:custody_v3	HTTP	SSL V3	
Replication	uddi-org:replication_v3	HTTP	SSL V3	Mutual authentication
Subscription	uddi-org:subscription_v3 uddi-org:subscriptionListener_v3	HTTP	SSL V3	
Value Set	uddi-org:valueSetValidation_v3 uddi-org:valueSetCaching_v3	HTTP		

9.6.3 Domain key generator tModels

It is recommended that an authorization policy for saving a root key generator tModel based on a domain key be established by the registry. This policy should include whether the registry requires a Signature element and what constitutes a valid signature for saving the domain key generator tModel..

9.6.4 Replication Policies

Registries that use the replication protocol defined in Chapter 7 *Inter-Node Operation* to replicate data use a single-master data model. The nodes in a registry using this replication protocol MUST enforce the recommended policy in Section 7.1 *Inter-Node Policy Assertions.* The registry SHOULD define policies detailing the topology for replication and the frequency of replication API calls, or acceptable latency for processing changeRecord elements.

These policies MAY be declared using the Replication Configuration Structure as described in Section 7.5 *Replication Configuration*. If the registry policy requires that the nodes adhere to the details in the registry Replication Configuration Structure, a policy SHOULD further detail management of the Replication Configuration Structure. The policy for the management of the Replication Configuration Structure SHOULD include details on the authorized publisher of the Replication Configuration Structure and it is RECOMMENDED that the authorized publisher use the dsig:Signature element to sign the Replication Configuration Structure for integrity. It is further RECOMMENDED that the registry distribute the Replication Configuration Structure in a manner that restricts access to the file to the operators of nodes in the registry. Changes to the replication topology as well as the number of nodes in a registry MAY be initiated through changes to the Replication Configuration Structure. The RECOMMENDED method for using the Replication Configuration Structure to add and remove nodes from a registry using replication is described in Sections 7.8 and 7.9.

The registry SHOULD define policies detailing the authentication method that is used to authorize access to the change record journal for each node in the registry. It is RECOMMENDED that SSL Version 3.0 with mutual authentication be implemented by each node in a registry using the replication API set as described in Section 7.5.5 *Security Configuration.*

The total set of replication policies SHOULD be documented using the Replication Configuration Structure in conjunction with human readable documentation that is distributed to the operators of the nodes in a registry using the replication protocol.

This section describes the registry and node policies that must be established surrounding the use of value sets in UDDI. Value sets are identifier and categorization systems that are referenced using keyedReferences and are applied to tModels, businessEntities, businessServices, and binding Templates by publishers. Inquirers can use value sets to enhance discovery.

9.6.5 Value sets

9.6.5.1 Recognizing value sets

Publishers of value set tModels SHOULD categorize those that are for category systems with the *categorization* value, for identifier systems with the *identifier* value, and for category group systems with the *categorizationGroup* value.

9.6.5.2 Unchecked value sets

An unchecked value set is one that allows unrestricted references to its values. A UDDI registry is REQUIRED to have a policy to differentiate between unchecked value sets and checked value sets. UDDI registries MUST allow unchecked value sets to be referred to in keyedReferences. tModels that represent unchecked value sets SHOULD be categorized with the *unchecked* value from the uddi-org:types category system.

9.6.5.3 Checked value sets

Published keyedReferences involving checked value sets are validated using a validation algorithm acceptable to the value set provider. The most common validation is that referenced values are part of a predefined set. Checking can often be accommodated by the node using a cached set of valid values. More complicated or contextual validation can be handled by external validation services. See Section 5.6 *Value Set API Set* for more information.

tModels that represent checked value sets MUST be categorized with the *checked* value from the uddi-org:types category system. If a value set tModel is categorized as checked, then in response to attempts to publish a keyedReference which uses the checked tModel, nodes MUST either perform the required validation, or return E_unsupported.

Validation algorithms that do no more than verify that referenced values are part of a set of approved values can often be hosted by a node if the value set provider agrees to allow this to happen. tModels for checked value sets that allow caching of valid values for this simple kind of validation SHOULD be categorized with the *cacheable* categorization from the uddi-org:types category system. Similarly, if a tModel for a checked value set does not support caching of its values for validation, it SHOULD be categorized with the *uncacheable* categorization by uddi-org:types category system.

A node may acquire the set of valid values for a cached checked value set through private arrangements or through the invocation of a get_allValidValues Web service. A node can similarly validate references to a checked value set through private arrangements or through the invocation of a validate_values Web service. The tModel for a checked value set that has a get_allValidValues or validate_values Web service SHOULD be categorized with a reference to the bindingTemplate for the get_allValidValues or validate_values Web service using the

uddi-org:validatedBy category system. The referred to bindingTemplate SHOULD contain a reference to all value set tModels to which the value set Web service applies in the tModelInstanceDetails element.

A node SHOULD flush a valid values cache that was previously obtained through invocation of a get_allValidValues service when it receives notice as the custodial node or through the replication stream that the tModel for the checked value set has been republished. The recommended technique for a provider of a cached checked value set to notify UDDI nodes of new valid values is to republish the tModel for the value set. Providers of cached value sets SHOULD NOT delete valid values from the value set or change the meaning of values as this adversely affects entities that previously referenced the value set and erodes the confidence in these references.

9.7 UDDI Policy Summary

The tables below summarize the information presented in this chapter for easy access.

9.7.1 UDDI Registry Policy Abstractions

The following table captures the policies that a registry MAY specify.

Policy Group	Policy Name	Policy Rule Description	PDP	Sections	Type
Policy Delegation	Registry Policy Delegation	The registry may allow nodes to define their own policies.	Registry	9.4.1 Registry Policy Delegation	Document
	Delegation of User registration	A registry defines if nodes may specify their own user registration	Registry	1.7 Introduction to Security	Document
	Delegation of Authorization	A registry defines if nodes may specify their own access control policy	Registry	1.7 Introduction to Security 9.4.8.1 Delegation of Authorization Policy	Document
	Delegation of Subscription Policy	The registry defines if nodes may define their own policies for subscription.	Registry	9.4.18 Registry Subscription Policy	Document
	Value set policy delegation policy	Value set policy delegated to node	Registry	9.4.19.1 Value Set Delegation Policy	Document
Keying	Registry General Keying Policy	The registry defines a policy for key format and key generation.	Registry	9.4.2 Registry General Keying Policy	Document
UDDI keying	Registry Key Generation	The registry defines a policy for whether and how a given node or publisher is allowed to register a key generator tModel.	Registry [may be delegated]	5.2.2 Publishing entities with publisher assigned keys	Document

Policy Group	Policy Name	Policy Rule Description	PDP	Sections	Type
	Registry Key Default	The registry defines what happens when a key is not supplied.	Registry [may be delegated]	9.4.3.3 Registry Key Default	Document
	Registry Support of UUIDKeys	The registry defines whether uuidKeys are accepted.	Registry	9.4.3.4 Registry Support of UUID Keys	Document
	Registry Root Key Generation	The registry defines whether or not affiliations are allowed and how key partitions are maintained.	Registry [may be delegated]	9.4.3.5 Root Key Generation	Document
UDDI Information and Access Control Policies	Registry Registration	The registry defines a policy for registration of users.	Registry [may be delegated]	1.7 Introduction to Security 9.4.4 UDDI Information and Access Control Policy	Document
	Registry Authorization	The registry defines a policy for Authorization of users.	Registry [may be delegated]	9.4.8 Registry Authorization Policy	Model
	Registry Data Integrity	The registry defines a policy for Data Integrity.	Registry [may be delegated]	9.4.10 Registry Data Integrity	Model
	Registry Persistent Data Confidentiality	The registry defines a policy for persistent Data Confidentiality (data in the data repository)	Registry [may be delegated]	9.4.12 Registry Data Confidentiality	Document
	Registry Audit	The registry defines a policy for Audit	Registry [may be delegated]	9.4.13 Registry Policy Audit	Document
	Registry Privacy	The registry defines a policy for Privacy.	Registry [may be delegated]	9.4.14 Registry Privacy Policy	Document
	Registry Extensibility	The registry defines a policy for extensibility	Registry	Appendix H	Model
	Registering Nodes in a registry	The registry defines how nodes are added and deleted from a registry.	Registry	7.8 Adding a Node to a Registry Using Replication	Document
APIs	Data Confidentiality for Inquiry	The registry defines a policy for Data Confidentiality for the inquiry API set.	Registry [may be delegated]	9.4.12 Registry Data Confidentiality	Model

Policy Group	Policy Name	Policy Rule Description	PDP	Sections	Type
	Authorization for Inquiry	The registry determines if authorization is required on the API set and how this is supplied.	Registry [may be delegated]	9.4.9 Modeling Authorization	Model
	Data Confidentiality for Publish	The registry defines a policy for Data Confidentiality for the publish API set.	Registry [may be delegated]	9.4.12 Registry Data Confidentiality	Model
	Authorization for Publish	The registry determines if authorization is required on the API set and how this is supplied.	Registry [may be delegated]	9.4.9 Modeling Authorization	Model
	Data Confidentiality for Subscription	The registry defines a policy for Data Confidentiality for the subscription API set.	Registry [may be delegated]	9.4.12 Registry Data Confidentiality	Model
	Authorization for subscription	The registry determines if authorization is required on the API set and how this is supplied.	Registry [may be delegated]	9.4.9 Modeling Authorization	Model
	Data Confidentiality for Replication	The registry defines a policy for Data Confidentiality for the replication API set.	Registry [may be delegated]	9.4.12 Registry Data Confidentiality	Model
	Authorization for replication	The registry determines if authorization is required on the API set and how this is supplied.	Registry [may be delegated]	9.4.9 Modeling Authorization	Model
	Data Confidentiality for Custody Transfer	The registry defines a policy for Data Confidentiality for the Custody and Ownership Transfer API set.	Registry [may be delegated]	9.4.12 Registry Data Confidentiality	Model
	Authorization for custody transfer	The registry determines if authorization is required on the API set and how this is supplied.	Registry [may be delegated]	9.4.9 Modeling Authorization	Model
	Data Confidentiality for External validations	The registry defines a policy for Data Confidentiality for the external validations API set.	Registry [may be delegated]	9.4.12 Registry Data Confidentiality	Model
	Authorization for external validations	The registry determines if authorization is required on the API set and how this is supplied.	Registry [may be delegated]	9.4.9 Modeling Authorization	Model

Policy Group	Policy Name	Policy Rule Description	PDP	Sections	Type
Time Policies	Clock Synchroniza-tion Policy	The registry may define how nodes in a registry synchronize their clocks.	Registry	9.4.15 Registry Clock Synchronization Policy	Document
User Policies	Publication Limits	A registry defines the amount of information that publishers are able to publish.	Registry [may be delegated]	9.4.6.2 Publication Limits	Document
	Person, Publisher and Owner	A registry defines the relationship between data and publishers.	Registry	9.4.6 Person, Publisher and Owner	Document
	Transfer of Ownership	A registry defines if data is able to be transferred between owners in the registry.	Registry	9.4.7 Transfer of Ownership	Document
Data Custody	Registry support for Data Custody	Registries must specify whether custody transfer is supported	Registry	9.4.17.1 Modeling custody transfer	Document
Replication	Replication support for Data Custody	A registry defines if replication of transfer is supported	Registry	9.4.17 Replication support for Custody Transfer	Document
	Registry Support for Replication	The registry defines if replication is supported	Registry	7.1 Inter-Node Policy Assertions	Model
	Single Master Replication	The registry defines if the Single master data model for replication is supported	Registry	7.1 Inter-Node Policy Assertions	Model
Subscription	Registry Support for Subscription	The registry defines if subscription is supported.	Registry [may be delegated]	5.5 Subscription API Set 9.4.18 Registry Subscription Policy	Model
	Registry limits for email subscriptions	The registry defines limits for the number of email subscription-related notification-based retries.	Registry [may be delegated]	9.4.18 Registry Subscription Policy	Document
	Registry support for filter criteria	The registry defines if the Inquiry APIs are available for use as filter criteria in a subscription	Registry [may be delegated]	9.4.18.1 Registry Limits	Model

Policy Group	Policy Name	Policy Rule Description	PDP	Sections	Type
	Registry conditions for subscription renewal	The registry defines conditions for subscription renewal	Registry [may be delegated]	9.4.18.1 Registry Limits	Document
	Registry limits on subscription volume	The registry defines the limit on the volume of data to be returned in subscription results.	Registry [may be delegated]	9.4.18.1 Registry Limits	Document
	Subscription Duration	The registry defines the duration of time in which a subscription remains active.	Registry [may be delegated]	9.4.18.2 Subscription Duration	Document
Value Set Policy	Checked value sets policy	Checked values sets supported	Registry	9.4.19 Registry Value Set Policies	Document
	Cache invalidation policy	Cache Invalidation Trigger	Registry	9.4.19.4 Cache Invalidation Policy.	Document
	Uncached checked value sets policy	Uncached value sets supported	Registry [delegated]	9.4.19.3 Uncached checked value sets policy	Document
	External validation Web service supported policy	External validation Web services supported	Registry [delegated]	9.4.19.5 External Validation Policies	Document
	Value set Web service discovery policy	Modeling policy for registering and discovering value set Web services		9.4.19 Registry Value Set Policies	Document
	Data Integrity/Data Confidentiality	A policy for certificate authorities is supported	Node	9.4.11 Registry Approved Certificate Authorities	Document

9.7.2 UDDI Node Policy Abstractions

Policy Group	Policy Name	Policy Rule Description	PDP	Sections	Type
Node Key Policies	Node Key Generation	If delegated, a node may define which publishers are allowed to register tModels.	Node	5.2.1 Publishing entities with node assigned keys	Document

Policy Group	Policy Name	Policy Rule Description	PDP	Sections	Type
	Node Publisher Generated Key Assertion	Each node must establish whether or not it will accept publisher generated keys.	Node	9.5.2 Node Publisher Generated Keys	Document
Node Information Policy	Node Message Limit	A node may limit the maximum size of a request message	Node	9.5.13 Node Element Limits	Model
	Node Registration	The node defines a Policy for registration of users.	Node	9.5.5 Node Registration and Authentication	Document
	Node Publication Limits	A node defines the amount of information that publishers are able to publish.	Node	9.5.6 Node Publication Limits	Document
	Disclaimers	A policy for contesting entries is supported	Node	9.5.7 Node Policy for Contesting Entries	Document
	Node Authentication	The node defines a policy for Authentication of its registered users. Mapping between identification and authorization	Node	1.7 Introduction to Security 9.5.5 Node Registration and Authentication	Model
	Node Authorization	The node defines a policy for Authorization of its users.	Node	9.5.4 Node Authorization Policy	Model
	Node Privacy Policy	A node defines the privacy policy for the operational information that it collects and maintains as a result of registration.	Node		Document
	Node Audit Policy	A node defines its local policy for audit	Node		Document
	Node Availability Policy	A node defines a policy for its service availability.	Node		Document
	Node Sort Order	Each node MUST specify the default sort order supported. A node MAY specify support for any optional additional sort orders. All sort orders are specified via use of a sortOrder tModel.	Node	9.5.9 Node Sort Order Policy	Document

Policy Group	Policy Name	Policy Rule Description	PDP	Sections	Type
Node APIs	Node use of Security APIs	The node defines if the criteria for identifying authorized publishers is via authInfo	Node	4.7 About Access control and the authInfo Element	Model
	Authorization for Inquiry APIs	AuthInfo is supported on the Inquiry APIs	Node	4.7 About Access control and the authInfo Element	Model
	Authorization for Publish APIs	AuthInfo is supported on the Publish APIs	Node	4.7 About Access control and the authInfo Element	Model
	Authorization for Custody APIs	AuthInfo is supported on the Custody and Ownership Transfer APIs	Node	4.7 About Access control and the authInfo Element	Model
	Authorization for Subscription APIs	AuthInfo is supported on the Subscription APIs	Node	4.7 About Access control and the authInfo Element	Model
	Authorization for Value Set APIs	AuthInfo is supported on the Value Set APIs	Node	4.7 About Access control and the authInfo Element	Model
	Data Integrity/Data Confidentiality	A policy for certificate authorities is supported	Node	9.5.11Node Approved Certificate Authorities	Document
Misc.	Node Element Limits	A node may limit the number of elements within the same language.	Node	9.5.13 Node Element Limits	Document
	Node Discovery URLs	A node may establish a policy on whether or not it generates Discovery URLs.	Node	9.5.15 Node discoveryURL Generation	Document
	Node HTTP Get Service	A node MAY specify if it supports an HTTP GET service for access to the XML representations of UDDI data structures	Node	9.5.14 Node HTTP GET Services	Document
	Node XML Encoding	A node MAY specify the character encoding (UTF-8 or UTF-16) it uses in response messages.	Node	4.2 XML Encoding Requirements 9.5.16 Node XML Encoding Policy	Document

10 Multi-Version Support

There are instances when a UDDI node may support multiple UDDI API versions that interact with a common set of UDDI data. A UDDI node MAY choose to support the Version 3 specification while continuing to allow users to perform Version 2 inquiry and publish API calls[37]. In such a configuration, a node MUST respond to an API with behavior according to the namespace from which the API originated. For example, a find_business call within the uddi-org:api_v2 namespace MUST behave according to the Version 2 specification, while a find_business call within the uddi-org:api_v3 namespace MUST behave according the Version 3 specification – regardless of the fact that these queries were issued on an identical dataset.

There are situations where this guiding principle is not sufficient to address differences in the behavior of existing APIs as well as entirely new APIs that may not exist in an earlier version. To help node implementers in these unclear situations, this chapter will review the special considerations to be taken into account when supporting a multi-versioned node. This chapter also covers the considerations of migrating earlier versions of UDDI data to the Version 3 data structures.

10.1 Entity Key Compatibility with Earlier Versions of UDDI

The V3 key format change has some important considerations for implementations that wish to simultaneously support several versions of the UDDI APIs. This section explores how to support a multi-versioned UDDI implementation with regard to entity keys.

10.1.1 Generating Keys From a Version 3 API Call

A UDDI registry that wishes to support both UDDI v2 and UDDI v3 interfaces is faced (among other issues) with the problem of needing to manifest to its v2 inquirers keys for entities that were created using UDDI v3 and thus do not natively possess keys acceptable to the UDDI v2 key format as is the case for a v3 domainKey. The manner in which a UDDI v2 key is associated with such a UDDI v3 entity is not normatively defined, and so may be carried out by any means desired so long (of course) as the same result is seen by all UDDI v2 inquirers at any node in the registry.

The following approach is straightforward and efficient, and is RECOMMENDED. In particular, since this approach is entirely algorithmic, no additional information need be communicated or conveyed for this purpose between the nodes of the registry beyond that which would normally be necessary in a UDDI-v3-only registry.

A registry may establish as part of its key management policy, use of a direct mapping algorithm for UUID keys. This mapping consists of what follows: V3 keys that are UUID keys are transformed to V2 keys by removing the "uddi:" prefix, and in the case of a key referring to a tModel, pre-pending "uuid:" to the UUID. All other V3 keys that are not UUID keys are hashed using the hashing algorithm described herein.

Let k3 be a UDDI v3 key.

1. take an MD5 hash of the concatenation of:
    a. the 16-byte sequence 14 e3 a2 b1 3b d8 4c f5 af a6 0d 14 1b f3 20 76

[37] The same principal holds true for version 1 API calls.

b. the normalized form of k3 (including the required "uddi:" prefix); see Section 4.4 *About uuidKeys* for the normalization process

2. modify the 16 octets output of the MD5 hash:
   a. change the four most significant bits of the seventh octet to '0011'
   b. change the two most significant bits of the ninth octet to '10'
3. format the 16 octets in the form of a UUID string; in making this interpretation, we rely on the specification of UUIDs as found in http://uddi.org/pubs/draft-leach-uuids-guids-01.txt:
   a. convert them into a hexadecimal string
   b. separate them into groups of 8, 4, 4, 4, and 12 hexadecimal digits with hyphens

Once this is done, one straightforwardly defines a UUID v2 key for the entity denoted by k3 in the normal UDDI v2 manner as appropriate for the type of that entity.

Some examples of V3 domainKeys that have been processed into UUID-based UDDI Version 1 and 2 keys using this algorithm are:

For the businessKey uddi:tempuri.org = 5de0d2b4-ce18-318a-a7fa-64692c42dc25

For the tModelKey uddi:tempuri.org:keygenerator = uuid:eabe885f-9de2-3924-bd41-9eff2ce52606

As shown in the examples above, keys for tModels in UDDI Version 1 and 2 were denoted with a prefix "uuid:" followed by the UUID. All other keys in UDDI Version 1 and 2 are in the format of a UUID without the prefix.

Note that while there exists a mapping between two keys, a client must use the appropriate key for the version being used. A Version 2 API must specify an entity with a Version 2 key and vice versa.

10.1.2 Generating Keys from a Version 2 API Call

In the case where an entity is saved in a multi-versioned registry using a Version 2 API, a different set of issues arise. Again, the manner in which this is accomplished is non-normative, but a correlation must be in place between the v2 and v3 entity. A recommended approach to this requirement is to generate a v3 key prior to generating a v2 key and then using the recommended hashing algorithm and/or a mapping algorithm to create a v2 key for the user. Again, because this approach is algorithmic, it introduces no replication issues.

As an example of the hashing algorithm, suppose a tModel is being published at a node of a registry that generates its V3 keys in the partition of the key generator "uddi:example.com:registry:sales:keygenerator". And suppose that the node assures uniqueness in the partition by generating monotonically increasing serial numbers.

When the publication is done with a v2 API, the node first generates a v3 key of the form "uddi:example.com:registry:sales:y" where y is the next serial number. It then generates a v2 using the MD5 hash discussed above.

As an example of the mapping algorithm, the node first generates a v3 UUID key of the form "uddi:3942306d-2438-492a-9b2a-185413b93673", it then generates the v2 key as "3942306d-2438-492a-9b2a-185413b93673" or "uuid:3942306d-2438-492a-9b2a-185413b93673" if the entity is a tModel.

Note that a publisher cannot perform "backward migration". In other words, one cannot correlate a v3 key that is proposed by a publisher and an existing v2 key. For example, if a business had published a businessEntity under v2 and then acquired a domainKey generator under v3, that publisher could not create a domainKey and have it be associated with the

existing businessKey created for the v2 entry. In this case, the publisher would have to delete the first entity and resave the entity with the new domainKey.

10.1.3 Migrating Version 2 keys to a Version 3 Registry

Migrating data containing v2 format keys is introduces a different set of issues. One needs to generate and add v3-format keys to match keys already in the data. Implementations are free to choose whatever algorithm they choose for this, but a correlation must be maintained between the existing v2 entity and its correlative v3 entity. Because the hashing solution discussed above is a one way hash, a different mapping is required to preserve the v2 key of the existing v2 entity while creating a valid v3 key.

A mapping can still be created algorithmically which does not require nodes to transmit additional information to one another as follows:

10.1.3.1 Within a Root Registry

During migration, to generate a v3 key corresponding to a v2 tModelKey, replace the "uuid:" prefix with the v3 prefix "uddi:". To generate the v3 key corresponding to any other type of key, prepend the v2 key with the v3 prefix "uddi:".

For example, the v2 tModelKey "**uuid**:68DE9E80-AD09-469D-8A37-088422BFBC36" would correspond to the v3 key "**uddi**:68DE9E80-AD09-469D-8A37-088422BFBC36" and the v2 businessKey "D2033110-3AAF-11D5-80DC-002035229C64" would correspond to the v3 key "**uddi**:D2033110-3AAF-11D5-80DC-002035229C64".

Note that this algorithm is transitive: after migration, given a v3 key, one can determine its v2 equivalent by applying this same pattern in reverse.

10.1.3.2 Within an Affiliate Registry

In order to insure that the affiliate registry's keys are unique in the context of other registries, the affiliate registry cannot migrate to v3 keys until it has established a key generator tModel with the root registry. By establishing a key generator, the affiliate registry can then migrate its keys, using the keyGenerator prefix as a basis for its keys.

For example, the affiliate might establish the key generator, "uddi:example.com:keygenerator". It would use this prefix when migrating the entirety of its v2 keys. For example, the v2 tModelKey "**uuid**:68DE9E80-AD09-469D-8A37-088422BFBC36" would correspond to the v3 key "**uddi:example.com:**68DE9E80-AD09-469D-8A37-088422BFBC36" and the v2 businessKey "D2033110-3AAF-11D5-80DC-002035229C64" would correspond to the v3 key "**uddi:example.com:**D2033110-3AAF-11D5-80DC-002035229C64".

Again, this algorithm is transitive: after migration, given a v3 key, one can determine its v2 equivalent by stripping the keyGenerator prefix and, in the case of tModel, appending "uuid:". It is important that all nodes in the affiliate registry are aware of the algorithm used during the migration such that a correlation is maintained.

10.1.4 Mapping v1/v2 Canonical tModel Keys to v3 Evolved Keys

Another exception to the algorithmic mapping is for the v1/v2 canonical tModel keys. In order to preserve these ubiquitous keys, there must be a direct mapping to the v1/v2 GUIDs documented in the v1/v2 specification. As such, nodes responding to a v2 request with one of the v3 canonical tModels must not generate a hashed key, but rather must correlate the v3 with an existing GUID. The list of those GUIDs and domainKeys are listed below and are also clearly noted in Chapter 11 Utility tModels and Conventions as evolved keys as opposed to derived keys.

V3 key	V1/V2 key
uddi:uddi.org:categorization:types	uuid:C1ACF26D-9672-4404-9D70-39B756E62AB4
uddi:uddi.org:categorization:general_keywords	uuid:A035A07C-F362-44dd-8F95-E2B134BF43B4
uddi:uddi.org:categorization:nodes	uuid:327A56F0-3299-4461-BC23-5CD513E95C55
uddi:uddi.org:relationships	uuid:807A2C6A-EE22-470D-ADC7-E0424A337C03
uddi:uddi.org:categorization:owningbusiness	uuid:4064C064-6D14-4F35-8953-9652106476A9
uddi:uddi.org:identifier:isreplacedby	uuid:E59AE320-77A5-11D5-B898-0004AC49CC1E
uddi:uddi.org:transport:http	uuid:68DE9E80-AD09-469D-8A37-088422BFBC36
uddi:uddi.org:transport:smtp	uuid:93335D49-3EFB-48A0-ACEA-EA102B60DDC6
uddi:uddi.org:transport:ftp	uuid:5FCF5CD0-629A-4C50-8B16-F94E9CF2A674
uddi:uddi.org:transport:fax	uuid:1A2B00BE-6E2C-42F5-875B-56F32686E0E7
uddi:uddi.org:transport:telephone	uuid:38E12427-5536-4260-A6F9-B5B530E63A07

10.1.4.1 Response for V2 inquires for evolved tModels

When a V2 find_tModel or get_tModelDetail inquiry for the V2 tModels evolved into the V3 specification as part of the response will be based on the tModel Structure in Chapter 11 of the V3 specification. The specific difference in the tModels when a V2 client queries a multi-version V3 registry for the evolved V2 tModels are that the tModel overviewURL and keyedReference elements in the categoryBag will be from the V3 specification.

10.2 Version 2 API Considerations

10.2.1 Multiple xml:lang attributes of the same language

In Versions 2 there could only be one name or description element with a given xml:lang. In Version 3 of UDDI there can be multiple names with the same language attribute. When this occurs the first name or description is considered the default for that language.

10.2.2 Error codes

Registries that support multiple versions of the UDDI APIs respond with error codes appropriate to the version of the APIs that are invoked. For example, when a v2 API is invoked, a registry MUST NOT respond with a NEW v3 error code. In this case the registry SHOULD use the E_fatalError code.

There are also a number of error codes from V2 that are deprecated in V3. A registry MAY NOT return these specific errors to a V2 API call if the V3 data model does not require that the error be returned in the corresponding V3 API. By example, if too many names are specified in a V2 Inquiry API sent to a V3 registry, the V3 registry may successfully process the request instead of returning the V2 error E_tooManyOptions.

10.2.3 Return of a dispositionReport

A dispositionReport on success is not returned by v3; rather an empty message is returned instead. The use of a dispositionReport in v3 is reserved to signal an error condition. A registry MUST return a dispositionReport to a v2 client to signal success consistent with the v2 specification.

10.2.4 Mapping Between URLType and useType attribute on accessPoint

The v3 specification no longer supports the attribute URLType on the accessPoint element. Rather, it uses the useType attribute. This necessitates a mapping between the values of the v2 URLType with the values of the v3 useType.

When a v3 node is responding to a v2 inquiry, the v2 URLType will always be returned as "other". When a v3 node is responding to a v2 publication, the returned v2 URLType will always be "other" regardless of the original URLType in the publication message; this means that the data returned from a successful v2 save will NOT match the data sent due to this data conversion. V3 inquiries that return data stored from a v2 publication will always return "endpoint" as the useType.

10.2.5 Supporting External Value Set Providers Across Versions

A situation may arise when there may be incongruent value sets of a checked taxonomy between two different versions. For example, the uddi:uddi.org:categorization:types taxonomy has new values in Version 3 that were not specified in Version 2. In such a situation, it is permissible for an inquiry API to return data that would be considered "invalid" given the valid values for that taxonomy.

In the case of external value sets, the same schema version specified in a save_xx API call, whose categoryBag or identifierBag references an externally validated value set, is used by the node in the associated validate_values API call performed in order to complete the save_xx request. If the needed validate_values API is not available at the required schema version, then the save_xx request results in the error E_unsupported. In such cases, the error text clearly indicates the cause of the problem.

10.2.6 Version 3 Schema Assessment

The UDDI v3 specification mandates use of schema assessment. (See Section 6.1.1.1 *Processing By XML Schema Assessment*.) It is RECOMMENDED that a multi-version registry perform schema assessment on earlier versioned APIs. It is further RECOMMENDED that a multi-version registry perform schema assessment according to the errata for XML Schema for conformance to RFC 3066 for the language type used by xml:lang (see http://www.w3.org/2001/05/xmlschema-errata#e2-25).

10.2.7 XML Encoding

Because UTF-16 is not supported in both UDDI Version 1 and 2, all response messages to UDDI Version 1 or 2 API calls are encoded in UTF-8.

10.2.8 Length Discrepancies

A number of fields are permitted to be longer in v3 than in prior versions of the UDDI specification. In the case when a v2 inquiry is made on a UDDI node which supports multiple versions, the node MAY return a field length longer than is specified in the v2 specification.

10.2.9 White Space Handling

The v3 specification mandates the usage of schema assessment with regard to the handling of white space, which differs slightly from the handling of white space in earlier versions. (See Section 6.1.1.1 *Processing By XML Schema Assessment*.) A multi-version node may return XML to an earlier versioned API with data that has been processed by v3 schema assessment.

10.3 Version 2 Inquiry API Considerations

10.3.1 keyedReference data

With Version 3, tModelKey elements MUST be specified for a keyedReference structure. Requests to the Version 3 namespace containing keyedReference structures without tModelKey elements will fail schema validation and be rejected. Requests in the version 2 namespace with empty or absent tModelKey elements MAY be processed by a multi-version node as a reference to the general_keyWords tModel. As this node behavior was optional in version 2, It is STRONGLY RECOMMENDED that all UDDI version 2 clients provide tModelKey attribute in keyedReference elements.

10.3.2 keyedReferenceGroup data

Because keyedReferenceGroups elements did not exist in Version 2, they will not be returned when a Version 2 API requests an entity that in fact has keyedReferenceGroup elements.

10.3.3 Multiple overviewDoc data

With Version 3, an entity may have multiple overviewDoc elements. If a Version 2 API queries such an entity, the first overviewDoc element will be returned according to its document order.

10.3.4 Multiple personName data

With Version 3, a contact may have multiple personName elements. If a Version 2 API queries such an entity, the first personName element will be returned according to its document order.

10.3.5 Service Projections

Because service projections are not available in UDDI Version 1, they never appear in the result set of a Version 1 find_business or find_service inquiry.

10.3.6 Sorting and Matching Behavior

The set of sorting and matching findQualifiers, as well as default sorting and matching behavior, for the Version 3 namespace has changed significantly. Please refer to the appropriate version-specific specifications for detailed explanations of sorting and matching semantics.

10.4 Version 2 Publish API Considerations

10.4.1 Data update semantics consistent with request namespace

If a publisher saves data using a Version 3 API call and then attempts to update that data by performing a Version 2 save_xx API call and passing the identical key, the entity will not preserve any Version 3 data that is not part of the Version 2 entity. As is true with all save_xx operations, a Version 2 save_xx operation performed on a Version 3 registry that supports it completely replaces the entity being saved. If the entity being replaced previously contained data not expressible in that prior version, it will no longer contain such data after a successful prior version save_xx operation.

For example, if the Version 3 entity contained a signature and was then re-saved with a Version 2 call, the signature would be lost. This same principle holds true for Version 1 API calls.

10.4.2 keyedReference data

With Version 3, tModelKey elements MUST be specified for a keyedReference structure contained within Publish API requests. Requests to the Version 3 namespace containing keyedReference structures without tModelKey elements will fail schema validation and be rejected.

When migrating Version 2 keyedReference data to a Version 3 node, publishers MUST construct tModelKey values for migrated data in a manner consistent with Section 10.1.4 *Mapping v1/v2 Canonical tModel Keys to v3 Evolved Keys*.

A keyedReference referring to the UDDI Types Category System with a keyValue attribute equal to "keyGenerator" is reserved exclusively for key generator tModels. Any attempt to use it in a V2 Publish API call will fail with E_valueNotAllowed returned.

10.5 Data Migration and Multi-version Runtime Considerations

10.5.1 Empty Containers – Enforcement of Schema Strictness

In Version 3, the schema was changed to no longer allow "empty containers" – XML wrapper tags that stored no data. Version 3 enforces a minOccurs=1.

Migration behavior. The modification of behavior introduced in v3 requires that a migration of the data from v2 to v3 must prune any XML structures that were saved with such a structure. Without such pruning, it is possible that a v3 API call might return XML that is not valid according to the v3 schema. This would hold true for both structures within the find_xx API calls as well as structures within the get_xx and save_xx API calls.

Runtime behavior. Similar considerations are present in the face of operating a v2/v3 multi-version registry whereby v2 clients may publish information that does not adhere to the v3

schema. At the time of processing the publish operation, a v2/v3 multi-version node will apply the same pruning used to migrate v2 data to v3.

Elements subject to migration and runtime pruning. The following structures need to be pruned either in the case of migration from v2 to v3, or at runtime by a v2/v3 multi-version registry:

- <addressLine> in <address>
- <bindingTemplate> in <bindingTemplates>
- <businessInfo> in <businessInfos>
- <businessService> in <businessServices>
- <contact> in <contacts>
- <description> or <overviewURL> in <overviewDoc>
- <findQualifier> in <findQualifiers>
- <fromKey> or <tokey> in <keysOwned>
- <instanceParms> or <overviewDoc> in <instanceDetails>
- <keyedReference> or <keyedReferenceGroup> in <categoryBag>
- <keyedReference> in <identifierBag>
- <relatedBusinessInfo> in <relatedBusinessInfos>
- <serviceInfo> in <serviceInfos>
- <tModelInfo> in <tModelInfos>
- <tModelInstanceInfo> in <tModelInstanceDetails>

10.5.2 Length Validation During v2/v3 Migration and During Runtime in a v2/v3 Multi-version Registry

Similarly, Version 3 introduces the notion of length validation – both minLength and maxLength -- within the schema. This change also affects data that is migrated from v2 to v3. The following elements now enforce such length validation:

- accessPoint[38]
- addressLine
- authInfo
- description
- discoveryURL
- email
- keyName

[38] The Version 2 Schema requires the accessPoint field, but the schema does not enforce a minLength. However, Version 2 errata clarified this behavior and enforced through the specification that an accessPoint of minLength one is required. Therefore, migrating accessPoint between v2 and v3 is not an issue.

- keyValue
- name
- personName[39]
- phone
- useType

When a client submits a value that exceeds the maxLength, the value will no longer be truncated by the node, but rather, the XML message will not pass validation. More importantly, in terms of the minLength requirement, if there are instances of v2 elements and attributes that are not of the required minLength, those v2 entities will need to be pruned during migration and at runtime, so that they will return XML that is valid according to the v3 schema.

As stated in Section 10.2.8, a multi-version v2/v3 registry responding to a v2 inquiry MAY return a field length longer than is specified in the v2 specification.

Example: Consider the following valid XML file from a v2 registry:

```
<businessDetail generic="2.0" operator="Microsoft UDDI Services"
        truncated="false" xmlns="urn:uddi-org:api_v2">
   <businessEntity businessKey="176a3131-0c20-45d1-b31d-efb4f61b8b14"
            operator="sample" authorizedName="sample">
        <discoveryURLs>
            <discoveryURL useType="businessEntity">
                http://sample/uddipublic/discovery.ashx?businessKey=176a3131...
            </discoveryURL>
        </discoveryURLs>
        <name>sample</name>
        <description xml:lang="en" />
        <contacts/>
   </businessEntity>
</businessDetail>
```

In order to be compatible with a v3 registry, the v2 XML response would have to be migrated or at runtime be returned as follows:

```
<businessDetail generic="2.0" operator="Microsoft UDDI Services"
        truncated="false" xmlns="urn:uddi-org:api_v2">
   <businessEntity businessKey="176a3131-0c20-45d1-b31d-efb4f61b8b14"
        operator="sample" authorizedName="sample">
        <discoveryURLs>
            <discoveryURL useType="businessEntity">
                http://sample/uddipublic/discovery.ashx?businessKey=176a3131...
            </discoveryURL>
        </discoveryURLs>
        <name>sample</name>
   </businessEntity>
</businessDetail>
```

Note that the <description> element and the <contacts/> element have been pruned.

10.6 Value sets with entity keys as valid values

There are value sets which use entity keys as valid values. Special handling of these values is necessary in a multi-version registry as specified in Section 11.1.9 UDDI "Entity Key Values" Category System.

[39] The Version 2 Schema requires personName and the specification allows it to be minLength=0. This presents a problem during migration. There is not a normative way to handle this migration issue; nodes may handle this situation as they choose.

They must be mapped as entity keys are everywhere else they are used. For example, if a V2 application saves a tModel that includes a keyedReference that uses the owningBusiness Value Set, the keyValue that will be used will be the V2 key of the appropriate businessEntity. If a V3 application retrieves this tModel then the keyValue returned MUST be the V3 key of the same businessEntity.

This behavior applies to the following set of tModels as well as any other tModel categorized using the entityKeyValues category system:

- uddi-org:validatedBy
- uddi-org:derivedFrom
- uddi-org:isReplacedBy
- uddi-org:owningBusiness

11 Utility tModels and Conventions

To facilitate consistency in Service Description (tModel) registration, and to provide a framework for their basic organization within UDDI registries, a set of tModels has been established for UDDI. This section describes this set of tModels that facilitate registration of common information and the services provided by the UDDI registry itself. Registration of these tModels is MANDATORY for all UDDI registries. Implementation of these tModels depends on the type of tModel and on the structure of the registry.

In addition to these "canonical" conventions and tModels, the UDDI Business Registry has established further conventions and tModels that registries MAY wish to adopt. See the UDDI Business Registry for additional useful tModel descriptions[40].

In the sections that follow a number of attributes are called out for each tModel described. These include the name of the tModel, the description of the tModel, its categorization(s), and its keys. Version 3 format keys are used when accessing a UDDI Version 3 registry using UDDI Version 3 APIs. When a node supports multiple versions of UDDI, Version 1 and 2 format keys are used for the tModels when the Version 3 registry is accessed with a prior version API.

There are two kinds of Version 1 and 2 format keys. There are those keys for tModels that are new in Version 3. The Version 1 and 2 format keys for these tModels can be derived algorithmically. See Section 10.1.1 *Generating Keys From a Version 3 API Call* for more information. tModels with this kind of V1 and V2 format key have the *Derived V1, V2 Format Key* attribute. tModels that existed in a prior version of UDDI have a V1 and V2 format key that is migrated following the algorithm described in Section 10.1.4 *Mapping v1/v2 Canonical tModel Keys to v3 Evolved Keys*. tModels with this kind of V1 and V2 format key have the *Evolved V1, V2 Format Key* attribute.

11.1 Canonical Category Systems, Identifier Systems and Relationship Systems

In UDDI, tModels are used to establish the existence of a variety of concepts and to point to their technical definitions. tModels that represent value sets such as category, identifier, and relationship systems are used to provide additional data to the UDDI core entities to facilitate discovery along a number of dimensions. This additional data is captured in keyedReferences that reside in categoryBags, identifierBags, or publisherAssertions. The tModelKey attributes in these keyedReferences refer to the value set that related to the concept or namespace being represented. The keyValues contain the actual values from that value set. In some cases keyNames are significant, such as for describing relationships and when using the general keywords value set. In all other cases, however, keyNames are used to provide a human readable version of what is in the keyValue.

tModels related to value sets can be checked or unchecked. keyValue references to unchecked value sets are never validated. Their use is unrestricted. keyValue references to checked value sets are either rejected out of hand (when the UDDI node does not support the referenced checked value set) or validated. Validation can occur internally by a node or by invoking an external validation Web service.

[40] Some of the tModels that appeared in prior versions of the UDDI specification are not considered normative and are therefore not part of this specification. The UDDI Business Registry, however, continues to support these tModels.

tModels related to value sets can also be placed out of service by marking them unvalidatable. When a new reference to a tModel that is marked unvalidatable is encountered, the reference is automatically rejected.

Registration of and support for the tModels that follow are MANDATORY for all UDDI registries. In addition to these tModels, the UDDI Business Registry has defined a number of common value sets, for example the NAICS and UNSPSC category systems, that UDDI registries MAY support. These are described in the overviewURLs for the UDDI Business Registry tModels.

11.1.1 UDDI Types Category System

11.1.1.1 Introduction

To distinguish among various types of concept, UDDI has established the Types category system. Publishers should categorize the tModels they publish using values from uddi-org:types to make them easy to find. The approach to categorization of tModels within the UDDI Type Category system is consistent with that used for categorizing other entities using other category systems. The categorization information for each tModel is added to the **<categoryBag>** elements in a **save_tModel** API. One or more **<keyedReference>** elements are added to the category bag to indicate the types of the tModel that is being registered. See Appendix F *Using Categorization* for more information.

11.1.1.2 Design Goals

The goal of the UDDI Types category system is to establish an unambiguous, simple UDDI-compatible category system that distinguishes the kinds of concepts that tModels can represent.

11.1.1.3 tModel Definition

Name:	uddi-org:types
Description:	UDDI Type Category System
UDDI Key (V3):	uddi:uddi.org:categorization:types
Evolved V1,V2 format key:	uuid:C1ACF26D-9672-4404-9D70-39B756E62AB4
Categorization:	categorization
Checked:	Yes

11.1.1.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:categorization:types">
  <name>uddi-org:types</name>
  <description>UDDI Type Category System</description>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#UDDITypes
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:categorization"
      keyValue="categorization"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:checked"
      keyValue="checked"
```

```
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:cacheable"
      keyValue="cacheable"
      tModelKey="uddi:uddi.org:categorization:types"/>
  </categoryBag>
</tModel>
```

11.1.1.4 Valid Values

Checking of references to this value set consists of ensuring that the keyValues are from the set of categories listed below. No contextual checks are performed unless otherwise specified for a given value.

The following constitute the value set for this category system. The valid values are those categories marked as being "allowed". These values are used in the keyValue attributes of keyedReference elements that are contained in categoryBag elements.

ID	Parent ID	Allowed	Description
tModel		No	These types are used for tModels
valueSet	tModel	Yes	Value set
identifier	valueSet	Yes	Identifier system
namespace	valueSet	Yes	Namespace
categorization	valueSet	Yes	Categorization system
postalAddress	categorization	Yes	Postal address system
categorizationGroup	tModel	Yes	Category group system
relationship	tModel	Yes	Relationship type system
specification	tModel	Yes	Specification for a Web service
xmlSpec	specification	Yes	Specification for a Web service using XML messages
soapSpec	xmlSpec	Yes	Specification for interaction with a Web service using SOAP messages
wsdlSpec	specification	Yes	Specification for a Web service described in WSDL
protocol	tModel	Yes	Protocol
transport	protocol	Yes	Wire/transport protocol
signatureComponent	tModel	Yes	Signature component
unvalidatable	tModel	Yes	Prevents a checked value set from being used
checked	tModel	Yes	Checked value set
unchecked	tModel	Yes	Unchecked value set
cacheable	tModel	Yes	Cacheable checked value set
uncacheable	tModel	Yes	Uncacheable checked value set
keyGenerator	tModel	Yes	Key generator (Note: A contextual check is performed as specified below

ID	Parent ID	Allowed	Description
			if this value is used)
findQualifier	tModel	Yes	Find qualifier
sortOrder	findQualifier	Yes	Sort order
useTypeDesignator	tModel	Yes	Designates a kind of usage for the pieces of data with which it is associated
bindingTemplate		No	These types are used for bindingTemplates
wsdlDeployment	bindingTemplate	Yes	bindingTemplate represents the WSDL deployment of a Web service

- tModel: The UDDI type category system is structured to allow for categorization of registry entries other than tModels. This key is the root of the branch of the category system that is intended for use in categorization of tModels within the UDDI registry. Categorization is not allowed with this key.
- valueSet: A valueSet is the parent branch for the identifier, namespace, and categorization values in this category system. A tModel categorized with this value indicates it can be referenced by some other value set tModel to indicate redefinition of purpose, derivation, extension or equivalence.
- identifier: An identifier tModel represents a specific set of values used to uniquely identify information. Identifier tModels are intended to be used in keyedReferences inside of identifierBags. For example, a Dun & Bradstreet D-U-N-S® Number uniquely identifies companies globally. The D-U-N-S® Number system is an identifier system.
- namespace: A namespace tModel represents a scoping constraint or domain for a set of information. In contrast to an identifier, a namespace does not have a predefined set of values within the domain, but acts to avoid collisions. It is similar to the namespace functionality used for XML. For example, the uddi-org:relationships tModel, which is used to assert relationships between businessEntity elements, is a namespace tModel.
- categorization: A categorization tModel is used for category systems within the UDDI registry. NAICS and UNSPSC are examples of categorization tModels.
- postalAddress: A postalAddress tModel is used to identify different forms of postal address within the UDDI registry. postalAddress tModels may be used with the address element to distinguish different forms of postal address.
- categorizationGroup: A categorizationGroup tModel is used to relate one or more category system tModels to one another so that they can be used in keyedReferenceGroups.
- relationship: A relationship tModel is used for relationship categorizations within the UDDI registry. relationship tModels are typically used in connection with publisher relationship assertions.
- specification: A specification tModel is used for tModels that define interactions with a Web service. These interactions typically include the definition of the set of requests and responses, or other types of interaction that are prescribed by the Web service. tModels describing XML, COM, CORBA, or any other Web services are specification tModels.

- xmlSpec: An xmlSpec tModel is a refinement of the specification tModel type. It is used to indicate that the interaction with the Web service is via XML. The UDDI API tModels are xmlSpec tModels.
- soapSpec: Further refining the xmlSpec tModel type, a soapSpec is used to indicate that the interaction with the Web service is via SOAP. The UDDI API tModels are soapSpec tModels, in addition to xmlSpec tModels.
- wsdlSpec: A tModel for a Web service described using WSDL is categorized as a wsdlSpec.
- protocol: A tModel describing a protocol of any sort.
- transport: A transport tModel is a specific type of protocol. HTTP, FTP, and SMTP are types of transport tModels.
- signatureComponent: A signature component is used to for cases where a single tModel can not represent a complete specification for a Web service. This is the case for specifications like RosettaNet, where implementation requires the composition of three tModels to be complete - a general tModel indicating RNIF, one for the specific PIP, and one for the error handling services. Each of these tModels would be of type signature component, in addition to any others as appropriate.
- unvalidatable: Used to mark a categorization or identifier tModel as unavailable for use by keyedReferences. A value set provider may mark its value set tModel *unvalidatable* if it wants to temporarily disallow its use. See Section 6.4 *Checked Value Set Validation* for more information.
- checked: Marking a tModel with this categorization asserts that it represents a value set or category group system whose use, through keyedReferences, may be checked. Registry, and possibly node policy determines when and how a checked value set is supported.
- unchecked: Marking a tModel with this categorization asserts that it represents a value set or category group system whose use, through keyedReferences, is not checked.
- cacheable: Marking a tModel with this categorization asserts that it represents a checked value set or category group system whose values may be cached for validation. The validation algorithm for a supported cacheable checked value set or category group system must rely solely upon matching references against the cached set of values.
- uncacheable: Marking a tModel with this categorization asserts that it represents a checked value set or category group system whose values must not be cached for validation. The validation algorithm for a supported uncacheable checked value set must be specified and associated with the tModel marked with this categorization and may consider contextual criteria involving the entity associated with the reference.
- keyGenerator: Marking a tModel with this categorization designates it as one whose tModelKey identifies a key generator partition that can be used by its owner to derive and assign other entity keys. This categorization is reserved for key generator tModels. Any attempt to use this categorization for something other than a key generator tModel will fail with E_valueNotAllowed returned.
- findQualifier: A findQualifier tModel is used as the value of a findQualifier element to indicate the type of processing to occur for the inquiry function in which it is included.
- sortOrder: A sort order tModel defines a collation sequence that can be used during inquiries to control ordering of the results.

- useTypeDesignator: A useTypeDesignator tModel is used to describe the way a piece of data should be interpreted. It is frequently used to extend the space of resource types found at a URI, such as access points, overview URLs, and discovery URLs. UDDI designates a set of common use types as simple strings; tModels of the useTypeDesignator type are used to describe others.
- bindingTemplate: This key is the root of the branch of the category system that is intended for use in categorization of bindingTemplates within the UDDI registry. Categorization is not allowed with this key.
- wsdlDeployment: A bindingTemplate categorized as a wsdlDeployment contains within its accessPoint the endpoint for a WSDL deployment document.

11.1.1.5 Example of Use

The following demonstrates how to categorize a tModel as representing a checked value set. The UDDI approximateMatch findQualifier tModel, for example, is categorized this way.

```
<tModel
   tModelKey="uddi:uddi.org:findqualifier:approximatematch">
  …
  <categoryBag>
    <keyedReference keyName="uddi-org:types:categorization"
      keyValue="findQualifier"
      tModelKey="uddi:uddi.org:categorization:types"/>
   </categoryBag>
</tModel>
```

11.1.2 General Keyword Category System

11.1.2.1 Introduction

Usually, category systems in UDDI are defined by registering a new tModel to represent the value set, but sometimes such formality is unnecessary. The UDDI General Keywords Category System provides a way of informally defining any number of unchecked value sets, each consisting of a namespace identifier and an associated set of category values. See Appendix F *Using Categorization* for more information.

11.1.2.2 Design Goals

Provide a simple, lightweight means for establishing and using unchecked UDDI category systems. Such value sets are generally fairly simple and often of interest only to a small number of people. Checked value sets must, and complex or broadly interesting value sets should be defined by registering a new tModel, which is the formal means of documenting the meaning and intended use of a value set.

11.1.2.3 tModel Definition

Name:	uddi-org:general_keywords
Description:	Category system consisting of namespace identifiers and the keywords associated with the namespaces
UDDI Key (V3):	uddi:uddi.org:categorization:general_keywords
Evolved V1,V2 format key:	uuid:A035A07C-F362-44dd-8F95-E2B134BF43B4
Categorization:	categorization

Checked: Yes

11.1.2.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:categorization:general_keywords">
  <name>uddi-org:general_keywords</name>
  <description>Category system consisting of namespace
         identifiers and the keywords associated with
         the namespaces.
  </description>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#GenKW
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:categorization"
      keyValue="categorization"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:checked"
      keyValue="checked"
      tModelKey="uddi:uddi.org:categorization:types"/>
  </categoryBag>
</tModel>
```

11.1.2.4 Valid Values

The general_keywords category system in UDDI behaves differently than do any of the other category systems. Like other category systems, the general_keyword category system is used in keyedReference elements to categorize the entities. Unlike other category systems, in the general_keyword category system both the keyName and the keyValue attributes of keyedReference elements are semantically meaningful and are required. The keyName identifies a particular value set and the keyValue specifies the value within that value set. With other category systems, the keyName plays no semantic role -- it is essentially commentary. This difference is reflected in the UDDI inquiry APIs: When a keyedReference containing a reference to the general_keywords category system appears in an inquiry, the keyNames are used.

Although UDDI requires only that keyName attributes be specified when used with the general_keywords, such keyNames SHOULD be URNs -- with what the W3C calls "an institutional commitment to persistence" -- in a domain name space you own. Following this convention will help avoid name collisions.

UDDI places no limitations on the value of keyValue attributes for keyedReferences that reference this tModel.

Checking for this category system consists of ensuring that keyName is not omitted or specified as the zero-length string; UDDI registries MUST fail save operations that contain keyedReferences that involve uddi-org:general_keywords and that do not specify a non-empty keyName.

11.1.2.5 Example of Use

In The Analytical Language of John Wilkins (translated from the Spanish El idioma analítico de John Wilkins by Lilia Graciela Vázquez; edited by Jan Frederik Solem with assistance from Bjørn Are Davidsen and Rolf Andersen) Jorge Luis Borges discusses the problems inherent to any system of classification. The "ambiguities, redundancies and deficiencies remind us of those which doctor Franz Kuhn attributes to a certain Chinese encyclopedia entitled Celestial Empire of Benevolent Knowledge. In its remote pages it is written that the animals are divided

into: (a) belonging to the emperor, (b) embalmed, (c) tame, (d) sucking pigs, (e) sirens, (f) fabulous, (g) stray dogs, (h) included in the present classification, (i) frenzied, (j) innumerable, (k) drawn with a very fine camelhair brush, (l) et cetera, (m) having just broken the water pitcher, (n) that from a long way off look like flies."

While this taxonomy has been widely referred to, it turns out that Borges probably made the whole thing up. Legitimate or bogus, the taxonomy certainly makes his point: "[I]t is clear that there is no classification of the Universe not being arbitrary and full of conjectures."

For some unknowable reason, Island Trading (islandtrading.example, a completely fictitious outfit) is organized internally using this category system, one division per category. (Division "m" is very small.) It wishes to categorize the business services it offers according to the division that offers it, and it wants to use the Celestial Empire taxonomy to do so. Since the category is only of interest to Island Trading, it is decided that the general_keyword approach will be used. "islandtrading.example:categorization:animals" is chosen to represent the taxonomy. This is the URN that is placed into the keyName attributes of keyedReferences that refer to this taxonomy. The Tame Division and the Fabulous Division both have catalog browsing business services. They appear as follows:

```
<businessServices>
  <businessService>
    <name>Island Trading Tame Animal Catalog Service</name>
    <description xml:lang="en">
       Search our Tame animals catalog on line
    </description>
    <bindingTemplates>
      <bindingTemplate>
        <accessPoint useType="endpoint">
           https://islandtrading.example/tame/catalog.html
        </accessPoint>
        <tModelInstanceDetails>
          <tModelInstanceInfo
            tModelKey="uddi:uddi.org:ubr:transport:http">
          </tModelInstanceInfo>
        </tModelInstanceDetails>
      </bindingTemplate>
    </bindingTemplates>
    <categoryBag>
      <keyedReference
         tModelKey="uddi:uddi.org:categorization:general_keywords"
         keyName="islandtrading.example:categorization:animals"
         keyValue="c"/>
      <keyedReference
         tModelKey="uddi:uddi.org:ubr:categorization:unspsc"
         keyName="UNSPSC: Livestock" keyValue="101015"/>
    </categoryBag>
  </businessService>
  <businessService>
    <name>
      Celestial Animals Fabulous Animal Books Catalog Service
    </name>
    <description xml:lang="en">
      Search our tame animals catalog on line
    </description>
    <bindingTemplates>
      <bindingTemplate>
        <accessPoint content="endpoint">
          https://islandtrading.example/fabulous/catalog.html
        </accessPoint>
        <tModelInstanceDetails>
          <tModelInstanceInfo
            tModelKey="uddi:uddi.org:ubr:transport:http">
          </tModelInstanceInfo>
        </tModelInstanceDetails>
      </bindingTemplate>
    </bindingTemplates>
    <categoryBag>
      <keyedReference
```

```
            tModelKey="uddi:uddi.org:categorization:general_keywords"
            keyName="islandtrading.example:categorization:animals"
            keyValue="f"/>
          <keyedReference
            tModelKey="uddi:uddi.org:ubr:categorization:unspsc"
            keyName="unspsc-org:UNSPSC: Picture or drawing or
                coloring books for children"
            keyValue="55-10-15-07"/>
        </categoryBag>
      </businessService>
    </businessServices>
```

11.1.3 UDDI Nodes Category System

11.1.3.1 Introduction

UDDI provides a mechanism that may be used by publishers to categorize businessEntity and tModel elements according to any number of category systems and by inquirers to discover entities so categorized. See Appendix F *Using Categorization* for more information.

This section defines a tModel used to categorize a businessEntity as representing a UDDI node in the registry in which the businessEntity appears. See Section 6.2.2.1 *Normative Modeling of Node Business Entity*.

11.1.3.2 Design Goals

Each UDDI registry can be comprised of a number of nodes. Each UDDI node has a special businessEntity associated with it, called its Node Business Entity. The businessService elements in this businessEntity represent Web services that relate to the node's role in the UDDI registry.

The uddi-org:nodes category system is designed to allow reliable discovery of the Node Business Entity structures for nodes in a UDDI registry so that UDDI clients can locate the businessService structures associated with the operation of the registry.

11.1.3.3 tModel Definition

Name:	uddi-org:nodes
Description:	Category system for identifying nodes of a registry
UDDI Key (V3):	uddi:uddi.org:categorization:nodes
Evolved V1,V2 format key:	uuid:327A56F0-3299-4461-BC23-5CD513E95C55
Categorization:	categorization
Checked:	Yes

11.1.3.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:categorization:nodes">
  <name>uddi-org:nodes</name>
  <description>Category system for identifying the nodes
              of a registry.
  </description>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#Nodes
```

```
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:categorization"
      keyValue="categorization"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:checked"
      keyValue="checked"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:uncacheable"
      keyValue="uncacheable"
      tModelKey="uddi:uddi.org:categorization:types"/>
  </categoryBag>
</tModel>
```

11.1.3.4 Valid Values

Checking of references to this value set consists of ensuring that the publisher is the UDDI node and the keyValue has the value "node". Each node allows the use of uddi-org:nodes only by itself, only on its own Node Business Entity, and only with the value of "node". This value is used in the keyValue attributes of keyedReference elements that are contained in categoryBag elements to locate the Node Business Entity elements in the registry.

11.1.3.5 Example of Use

Consolidated Holdings (consolidatedholdings.example, a fictitious company) has become a node in a UDDI registry. As it sets itself up, it categorizes its Node Business Entity with the uddi:-org:nodes category system in the following way:

```
<businessEntity businessKey="uddi:consolidatedholdings.example"
...
  <categoryBag>
    <!-- Identify this businessEntity as a Node Business Entity -->
    <keyedReference keyName="uddi-org:nodes:Consolidated Holdings"
      keyValue="node"
      tModelKey="uddi:uddi.org:categorization:nodes"/>
  </categoryBag>
...
</businessEntity>
```

11.1.4 UDDI Relationships System

11.1.4.1 Introduction

UDDI provides a mechanism that may be used by publishers to assert relationships between businessEntity structures they publish and other businessEntity structures according to any number of relationship type schemes. See Appendix A *Relationships and Publisher Assertions* for more information. This section defines a tModel representing a relationship type system for use in describing the way businessEntity structures relate to one another.

11.1.4.2 Design Goals

While UDDI provides for any number of relationship type system to be used in relating businessEntity structures to one another, it is useful to define a "starter set" of relationship types that publishers may use without needing to define their own. The uddi-org:relationships relationship type system is such a starter set that covers a number of basic relationships. All three attributes are significant in keyedReferences that describe relationship types. The keyValue attributes should contain well known but somewhat broad versions of the relationship type, like those described in this relationship type set. The keyName attributes should be used to more explicitly type the relationship.

11.1.4.3 tModel Definition

Name:	uddi-org:relationships
Description:	Basic types of businessEntity relationships
UDDI Key (V3):	uddi:uddi.org:relationships
Evolved V1,V2 format key:	uuid:807A2C6A-EE22-470D-ADC7-E0424A337C03
Categorization:	relationship
Checked:	No

11.1.4.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:relationships">
  <name>uddi-org:relationships</name>
  <description>Basic types of business relationships
  </description>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#Relationships
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:categorization"
      keyValue="categorization"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:unchecked"
      keyValue="unchecked"
      tModelKey="uddi:uddi.org:categorization:types"/>
   </categoryBag>
</tModel>
```

11.1.4.4 Valid Values

The following constitute the value set for this relationship type system. The valid values are those types marked as being allowed. These values are used in the keyValue attributes of keyedReferences that are associated with publisherAssertions. The keyName attributes are also significant when used in publisherAssertions and can be used to more explicitly describe the relationship.

ID	ParentID	Allowed	Description
Relationship		No	The root of the relationships type system.
peer-peer	Relationship	Yes	Indicates that the two businessEntity structures are related as peers.
parent-child	Relationship	Yes	Indicates that the businessEntity referred to by the fromKey is in some sense the parent of the businessEntity referred to by the toKey.
identity	Relationship	Yes	Indicates that the businessEntity referred to by the fromKey represents the same business or organization as the businessEntity referred to by the toKey.

11.1.4.5 Example of Use

Tokyo Traders has a subsidiary, Chiba Traders, and wishes to assert that Chiba Traders is, indeed, its subsidiary. It wishes to use the uddi-org:relationships type system to assert that the businessEntity for Tokyo Traders is related via a parent-child subsidiary relationship to Chiba Traders. To do so it sends an add_publisherAssertions API to the node at which it published its businessEntity. The new assertion contained in the API looks as follows:

```
<publisherAssertion>
  <!-- Specify Tokyo Traders' businessKey as fromKey-->
  <fromKey>
    uddi:tokyotraders.example:business
  </fromKey>
  <!-- Specify Chiba Traders businessKey as toKey-->
  <toKey>
    uddi:chibatraders.example:business
  </toKey>
  <!--Specify a subsidiary relationship using uddi-org:relationships -->
  <keyedReference keyName="subsidiary"
    keyValue="parent-child"
    tModelKey="uddi:uddi.org:relationships"/>
</publisherAssertion>
```

In the example above, the keyName is used to qualify the parent-child relationship.

Once Tokyo Traders has added this assertion and Chiba Traders has done the same, a relationship is formed. The find_relatedBusinesses API may then be used to, for example, find Tokyo Traders' subsidiaries. The result would include Chiba Traders.

Note: A relationship between two businessEntity structures will be formed using the publisherAssertion mechanism only if the publisher of each of the businessEntity structures

involved asserts an identical publisherAssertion. In particular, this means that the keyedReferences must match exactly on keyName, keyValue, and tModelKey.

11.1.5 UDDI "Owning Business" Category System

The owningBusiness tModel represents a category system that may be used to locate the businessEntity associated with the publisher of a tModel.

11.1.5.1 Design Goals

It is often desirable to be able to discover the business entity that represents the publisher of a given tModel. When choosing among similar Web service definitions, for example, it is useful to be able to determine that one of them is published by a known organization. For most UDDI entities this can be deduced by inspecting the containment hierarchy of the entity to its root businessEntity. For tModels, the UDDI owningBusiness category system fills this need by allowing tModels to point to the businessEntity of their publisher.

11.1.5.2 tModel Definition

Name:	uddi-org:owningBusiness
Description:	Category system used to point to the businessEntity associated with the publisher of the tModel
UDDI Key (V3):	uddi:uddi.org:categorization:owningbusiness
Evolved V1,V2 format key:	uuid:4064c064-6d14-4f35-8953-9652106476a9
Categorization:	categorization
Checked:	Yes

11.1.5.2.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:categorization:owningbusiness">
  <name>uddi-org:owningBusiness_v3</name>
  <description>Category system used to point to the businessEntity
              associated with the publisher of the tModel.
  </description>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#owningBusiness
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:categorization"
      keyValue="categorization"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:checked"
      keyValue="checked"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:uncacheable"
      keyValue="uncacheable"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="entityKeyValues"
      keyValue="businessKey"
      tModelKey="uddi:uddi.org:categorization:entitykeyvalues"/>
   </categoryBag>
</tModel>
```

11.1.5.3 Valid Values

The value set of this value set is the set of businessKeys. The content of keyValue in keyedReferences that refers to this tModel must be a businessKey. The keyValue is used to specify that the businessEntity whose businessKey is the keyValue in a keyedReference "owns" the tagged tModel. The entity tagged must be a tModel, the referred-to businessEntity must exist, and it must have been published by the same publisher.

11.1.5.4 Example of Use

The My API specification was published by *some business*. To indicate that this is so, the My API specification tModel has a keyedReference in its categoryBag that uses uddi-org:owningBusiness to point to the *some business* businessEntity.

```
<tModel tModelKey="uddi:some.business.example:myapispecification">
  <name>some-business-example:MyAPI</name>
  <categoryBag>
    <keyedReference keyName="owningBusiness:someBusiness"
      keyValue="uddi:some.business.example:business"
      tModelKey="uddi:uddi.org:categorization:owningbusiness"/>
  </categoryBag>
  ...
</tModel>
```

In this example, the keyName field serves to help readability but has no further meaning.

11.1.6 UDDI "Is Replaced By" Identifier System

UDDI provides a mechanism that may be used by publishers to tag their businessEntities and tModels with information that identifies them according to any number of identification systems. This tModel represents an identifier system that may be used to identify the tModel or businessEntity that logically replaces the tModel or businessEntity in which it is used. This version of the isReplacedBy identifier system replaces the prior version of this identifier system by providing a means for referring to replacement entities that have version 3 format keys.

11.1.6.1 Design Goals

It is often desirable to gracefully retire a tModel in favor of a replacement. For example, when a Web service definition is replaced by an incompatible version, the publisher of the specification may wish to leave the tModel for the existing definition in place so that existing uses will not be disturbed, while at the same time making it clear that there is a replacement available. The UDDI isReplacedBy identifier system, coupled with the behavior of UDDI with respect to obsolete tModels, fills this need by allowing the obsolete tModel to point to its replacement. See Section 5.2.11 *delete_tModel*.

The isReplacedBy identifier system exists in prior versions of UDDI. keyedReferences that refer to this original isReplacedBy identifier system contain entity keys in the version 1 and 2 formats (as UUIDs with the uuid or no scheme prefix). When accessed using a prior version API in a multi-version registry, the older isReplacedBy identifier system yields valid references to businessEntity or tModel keys that are in the format of the prior version, and thus remain valid. When viewed using the version 3 UDDI API these same references to the earlier isReplacedBy identifier system contain invalid version 3 format keys. A new version of this identifier system is required to be able to reference the set of values defined by version 3 format keys.

11.1.6.2 tModel Definition

Name: uddi-org:isReplacedBy

Description:	An identifier system used to point to the entity, using UDDI keys, that is the logical replacement for the one in which isReplacedBy is used.
UDDI Key (V3):	uddi:uddi.org:identifier:isreplacedby
Evolved V1,V2 format key:	uuid:e59ae320-77a5-11d5-b898-0004ac49cc1e
Categorization:	identifier
Checked:	Yes

11.1.6.2.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:identifier:isreplacedby">
  <name>uddi-org:isReplacedBy</name>
  <description>Identifier system used to point to the UDDI entity,
              using UDDI keys, that is the logical replacement
              for the one in which isReplacedBy is used.
  </description>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#IsReplacedBy
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:identifier"
      keyValue="identifier"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:checked"
      keyValue="checked"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:uncacheable"
      keyValue="uncacheable"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="entityKeyValues"
      keyValue="businessKey"
      tModelKey="uddi:uddi.org:categorization:entitykeyvalues"/>
    <keyedReference keyName="entityKeyValues"
      keyValue="tModelKey"
      tModelKey="uddi:uddi.org:categorization:entitykeyvalues"/>
   </categoryBag>
</tModel>
```

11.1.6.3 Valid Values

The keyValues in keyedReferences that refer to this tModel must be tModelKeys or businessKeys. Such a keyValue specifies the entity that is the replacement for the entity in which the keyedReference appears. The above and further validation requirements are as follows:

a. In the case where a reference is made from an obsolete business entity the following validation rules apply:

   1. reference to a new business entity; this is a valid operation
   2. reference to self; this is invalid
   3. reference to a service, binding or tModel; this is an invalid operation given that the entity being pointed to must be a business

4. reference to another publisher's business; this is a valid operation; no ownership check is made
5. reference to another publisher's service, binding or tModel; this is an invalid operation because of a.3 above
6. reference to invalid keys; this is an invalid operation; a key must be valid.

b. In the case where a reference is made from an obsolete tModel the following validation rules apply:

1. reference to a new tModel; this is a valid operation
2. reference to self; this is invalid
3. reference to a service, binding or business; this is an invalid operation given that the entity being pointed to must be a tModel
4. reference to another publisher's tModel; this is a valid operation; no ownership check is made
5. reference to another publisher's service, binding or business; this is an invalid operation because of b.3 above
6. reference to invalid keys; this is an invalid operation; a key must be valid.
7. reference to a hidden tModel; this is a valid operation

c. Adding isReplacedBy to a service's or binding's category bag: this is a semantically wrong operation and will be rejected.

When returning an error encountered in the above, E_invalidValue will be returned to indicate that a value that was passed in a keyValue attribute did not pass validation.

While this validation is intended at save time, references to replacing business entities may become invalid if (A) the business is deleted and (B) in V3 the business is deleted and then the key is re-used for a different entity. As such, in a replicating registry, nodes processing changeRecords related to business entities or tModels that refer to (now) invalid or missing business or tModels entity keys respectively, MUST NOT raise replication errors.

11.1.6.4 Example of Use

In the example below, the UDDI Version 2 uddi-org:publication_v2 tModel has been replaced by the uddi-org:publication_v3 tModel. To indicate this, the uddi-org:isReplacedBy identifier system is used to point the Version 2 uddi-org:publication_v2 tModel to the uddi-org:publication_v3 tModel. To do this the uddi-org:publication_v2 tModel has a keyedReference added to its identifierBag, as follows:

```
<tModel tModelKey="uuid:A2F33B65-2D66-4088-ABC7-914D0E05EB9E">
  ...
  <name>uddi-org:publication_v2</name>
  ...
  <identifierBag>
    <!-- Use uddi-org:IsReplacedBy to indicate that the
         uddi V2 publication tModel is logically replaced
         by the V3 publication tModel. -->
    <keyedReference keyName="isReplacedBy:publication_v3"
      keyValue="uuid:72ade754-c6cc-315b-b014-7c94791fe15c"
      tModelKey="uuid:e59ae320-77a5-11d5-b898-0004ac49cc1e"/>
  </identifierBag>
  ...
</tModel>
```

Here the keyName attribute is commentary serving to help readability. The keyValue specifies which tModel replaces this one -- the version 3 publication tModel in this case. And the

tModelKey specifies that the keyedReference is using the uddi-org:isReplacedBy identifier system..

11.1.7 UDDI "Validated By" Category System

This tModel represents a category system that is used to point a tModel representing a checked value set to the bindingTemplate for a value set caching or value set validation Web service.

11.1.7.1 Design Goals

One of the concepts that tModels can represent is a checked value set. A checked value set is one whose use is monitored by a validation algorithm. There are two types of validation algorithms: simple checking of referenced values against a pre-defined set of allowable values, and any other kind of validation. UDDI provides the Value Set API set (see Section 5.6 *Value Set API Set)* to acquire the set of allowable values or execute an external validation algorithm.

A validation algorithm for a checked value set can be acquired by nodes privately, or can be obtained through normal UDDI discovery. The validatedBy category system facilitates discovery of the value set caching or value set validation Web service for a checked value set tModel by pointing to the bindingTemplate for the Web service.

For the Web service to be useful, it must recognize any and all checked value sets that it is expected to be associated with. The recommended way for doing so is to place the tModels for the checked value sets it supports in the tModelInstanceDetails of the bindingTemplate for the Web service. Registry policy may require that providers of the Web service recognize value sets supported using this technique.

11.1.7.2 tModel Definition

Name:	uddi-org:validatedBy
Description:	A category system used to point a value set or category group system tModel to associated value set Web service implementations.
UDDI Key (V3):	uddi:uddi.org:categorization:validatedby
Derived V1,V2 format key:	uuid:25b22e3e-3dfa-3024-b02a-3438b9050b59
Categorization:	categorization
Checked:	Yes

11.1.7.2.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:categorization:validatedby">
  <name>uddi-org:validatedBy</name>
  <description>Category system used to point a value set or category
               group system tModel to associated value set Web service
               implementations.
  </description>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#validatedBy
    </overviewURL>
  </overviewDoc>
  <categoryBag>
```

```
    <keyedReference keyName="uddi-org:types:categorization"
      keyValue="categorization"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:checked"
      keyValue="checked"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:uncacheable"
      keyValue="uncacheable"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="entityKeyValues"
      keyValue="bindingKey"
      tModelKey="uddi:uddi.org:categorization:entitykeyvalues"/>
  </categoryBag>
</tModel>
```

11.1.7.3 Valid Values

The keyValues in keyedReferences that refer to this tModel must be bindingKeys. Such a keyValue SHOULD reference a bindingTemplate that specifies a Web service that implements a value set caching or value set validation API and which SHOULD reference the value set tModel so categorized with this category system. No other contextual checks are performed.

11.1.7.4 Example of Use

In the example below, an example checked value set tModel uses the validatedBy category system to refer to the bindingTemplate of a get_allValidValues Web service that is prepared to yield the set of valid values for the value set. The referenced bindingTemplate is shown below the tModel:

```
<tModel tModelKey="uddi:anexample:mycheckedvalueset">
  ...
  <categoryBag>
    <keyedReference keyName="uddi-org:types:categorization"
      keyValue="categorization"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:checked"
      keyValue="checked"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:cacheable"
      keyValue="cacheable"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <!-- Use uddi-org:validatedBy to point to the binding
         Template for the Web service that implements the
         get_allValidValues API. -->
    <keyedReference keyName="validatedBy:myCheckedValueSet:values"
      keyValue="uddi:anexample:mycheckedvalueset:values"
      tModelKey="uddi:uddi.org:identifier:validatedby"/>
  </categoryBag>
  ...
</tModel>
```

The referenced bindingTemplate for the get_allValidValues Web service is shown below. Note that the bindingTemplate references the value set tModel above, indicating it can yield valid values for that value set.

```
<bindingTemplate bindingKey="uddi:anexample:mycheckedvalueset:values"
serviceKey="…">
  <description>Web service to retrieve valid values for myCheckedValueSet
  </description>
  <accessPoint useType="endpoint">
     http://URL_of_service
  </accessPoint>
  <tModelInstanceDetails>
    <tModelInstanceInfo
      tModelKey="uddi:uddi.org:v3_valuesetcaching" />
    <tModelInstanceInfo
      tModelKey="uddi:anexample:mycheckedvalueset" />
```

```
    </tModelInstanceDetails>
  </bindingTemplate>
```

11.1.8 UDDI "Derived From" Category System

11.1.8.1 Introduction

UDDI provides a mechanism that may be used by publishers to categorize UDDI entities according to any number of category systems. See Appendix F *Using Categorization* for more information. This section defines a tModel used to associate a tModel, frequently a category system, with some other tModel, frequently the value set of some other category system, for the purpose of extension or redefinition of purpose.

11.1.8.2 Design Goals

Most value sets are used with some purpose in mind. To avoid ambiguity in publisher and inquirer intent it is not uncommon for this purpose to be explicitly associated with the value set in its tModel. The IS0 3166 geographic category system, for example, has the purpose *service offering area*.

Similarly, the UDDI API is comprised of a fixed set of programming interfaces and structures. UDDI registries can extend the UDDI API through schema derivation, to offer additional functionality.

The Derived From category system exists to allow tModels to refer to the tModels that they extend in some way. Value set values can be re-used by referring a derived value set tModel to the values in some other value set tModel. The reason for reuse can be for assigning another purpose to the set of values, for extending the set of values, for associating one set of values with another, or for some other kind of derivation.

Specification tModels that extend some other specification tModel can similarly use this category system to refer to the tModels they extend, providing end users with knowledge about the full scope of the API.

11.1.8.3 tModel Definition

Name:	uddi-org:derivedFrom
Description:	Category system for referring tModels to other tModels for the purpose of reuse
UDDI Key (V3):	uddi:uddi.org:categorization:derivedfrom
Derived V1,V2 format key:	uuid:5678dd4f-f95d-35f9-9ea6-f79a7dd64656
Categorization:	categorization
Checked:	Yes

11.1.8.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:categorization:derivedfrom">
  <name>uddi-org:derivedFrom</name>
  <description>Category system for referring tModels to other
               tModels for the purpose of reuse.
  </description>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#DerivedFrom
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:categorization"
      keyValue="categorization"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:checked"
      keyValue="checked"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:uncacheable"
      keyValue="uncacheable"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="entityKeyValues"
      keyValue="tModelKey"
      tModelKey="uddi:uddi.org:categorization:entitykeyvalues"/>
   </categoryBag>
</tModel>
```

11.1.8.4 Valid Values

The keyValue attribute in a keyedReference element that references this tModel within a categoryBag must be some other tModelKey in the UDDI registry. For value set derivations the tModel that is referred to contain the root values for the derived value set. A tModel for a derived value set is not automatically checked if the referred to value set is checked. The derived value set must itself go through the registry's process for making the derived value set checked.

11.1.8.5 Example of Use

Assume that the ISO 3166 Geographic system is a checked value set used to characterize where a business offers its Web services. Its tModel, after going through the registry's process for obtaining a checked value set looks like this:

```
<tModel tModelKey="uddi:uddi.org:ubr:categorization:iso3166">
  <name>ubr-uddi-org:iso-ch:3166-1999</name>
  <categoryBag>
    <!--Specify that this is a checked value set tModel by
        giving it "categorization" and "checked" values
        under the uddi-org:types category system -->
    <keyedReference keyName="uddi-org:types:categorization"
      keyValue="categorization"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:checked"
      keyValue="checked"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:cacheable"
      keyValue="cacheable"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <!--Further specify that this value set can be extended
          through derivation -->
    <keyedReference keyName="uddi-org:types:valueSet"
      keyValue="valueSet"
      tModelKey="uddi:uddi.org:categorization:types"/>
  </categoryBag>
</tModel>
```

To use the ISO 3166 values for the purpose of describing geographic location, a derived tModel is created with the new purpose. Note that even though the ISO 3166 value set tModel above is checked, meaning references to its values are validated, the derived tModel for describing the geographic location is not necessarily checked itself. To actually be checked, the provider of the validation algorithm must agree to check the values associated with the derived value set.

```
<tModel
  tModelKey="uddi:uddi.org:ubr:categorization:iso3166:
      business_location">
  <name>ubr-uddi-org:iso-ch:3166-1999:business_location</name>
  <categoryBag>
    <!--Derive values from the iso3166 value set -->
    <keyedReference
      keyName="derivedFrom:ubr-uddi-org:iso-ch:3166-1999"
      keyValue="uddi:uddi.org:ubr:categorization:iso3166"
      tModelKey="uddi:uddi.org:categorization:derivedfrom"/>
    <!--Specify that this is a checked value set tModel by
        categorizing it as "categorization"
        under the uddi-org:types category system -->
    <keyedReference keyName="uddi-org:types"
      keyValue="categorization"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types"
      keyValue="unchecked"
      tModelKey="uddi:uddi.org:categorization:types"/>
  </categoryBag>
</tModel>
```

A businessEntity that is categorized with geographic information with both intended purposes might look like this:

```
<businessEntity businessKey="uddi:a_business_key"
...
  <categoryBag>
    <!-- Categorize this businessEntity with original
         geographical purpose -->
    <keyedReference
      keyName="geoServiceArea:California, USA"
      keyValue="US-CA"
      tModelKey="uddi:uddi.org:ubr:categorization:iso3166"/>
    <!-- Categorize this businessEntity with derived
         geographical purpose -->
    <keyedReference keyName="geoLocation:California, USA"
      keyValue="US-CA"
      tModelKey="uddi:uddi.org:ubr:categorization:iso3166:
                 business_location"/>
  </categoryBag>
...
</businessEntity>
```

For an example on using the derivedFrom category system with a specification extension, see Appendix H *Extensibility*.

11.1.9 UDDI "Entity Key Values" Category System

The entity key values tModel represents a category system that may be used to indicate that a value set uses entity keys for its valid values.

11.1.9.1 Design Goals

There are several value sets in UDDI that have entity keys as valid values, and other such value sets may be defined. Special handling of the keyValue values is required for these value sets to ensure that they can be used by applications using any version of the UDDI API. By categorizing a value set with this tModel the publisher of the value set indicates that entity keys form the valid values of the value set. This allows a UDDI implementation to map entity keys between versions as is done with all other uses of entity keys.

If keys of only one type are valid for a particular value set, then that value set should have a single keyedReference relating to this tModel and the keyValue should contain the type of entity key that is valid, for example "tModelKey". If multiple types of key are valid, as in the case of uddi-org:isReplacedBy, then multiple keyedReferences can be used, one for each type of key. If any type of key is valid then a single keyedReference should be used with a keyValue of "entityKey".

A value set categorized with this tModel SHOULD be treated as an internally checked value set, whether or not it is also categorized as checked.

If the entity key supplied as the keyValue in a keyedReference relating to such a value set is not a valid entity key, or is the key of an entity of a type not supported by the particular value set, then the error E_invalidValue MAY be returned.

Value sets may require additional validation, and this additional validation MAY be performed before or after the validation of the key itself, therefore a different error MAY be returned if one of these additional validation steps fails before the validation of the key itself.

If an entityKeyValue value set is updated to remove all of the keyedReference elements referring to the "Entity Key Values" category system, a normative mapping behavior to update the keyValue of any existing references to the entityKeyValue value set is unspecified. Any new references or updates to existing references using keyedReference elements pointing to the tModel that formerly represented the entityKeyValue value set will be treated as a normal value set, where the keyValue is a string.

Further, if a tModel is updated to add at least one keyedReference element referring to the "Entity Key Values" category system, a normative mapping behavior to update the keyValue of any existing references to the entityKeyValue value set is unspecified. Any new references or updates to existing references using keyedReference elements pointing to the tModel that formerly represented the value set will be case folded and validated as an entityKeyValue value set, where the keyValue is verified to be an existing and appropriate entityKey.

In inquiry, the treatment of the keyValue is determined by the state of the value set tModel at the time of the inquiry. If the keyValue in an inquiry is contained in a keyedReference referring to the "Entity Key Value" set tModel, the keyValue must be case folded as part of the inquiry.

11.1.9.2 tModel Definition

Name:	uddi-org:entityKeyValues
Description:	Category system used to declare that a value set uses entity keys as valid values
UDDI Key (V3):	uddi:uddi.org:categorization:entitykeyvalues

Derived V1,V2 format key: uuid:916b87bf-0756-3919-8eae-97dfa325e5a4

Categorization: categorization

Checked: Yes

11.1.9.2.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:categorization:entitykeyvalues">
    <name>uddi-org:entityKeyValues</name>
    <description>Category system used to declare that a value set
                uses entity keys as valid values.
     </description>
     <overviewDoc>
         <overviewURL useType="text">
             http://uddi.org/pubs/uddi_v3.htm#entityKeyValues
          </overviewURL>
     </overviewDoc>
     <categoryBag>
        <keyedReference keyName="uddi-org:types:categorization"
          keyValue="categorization"
          tModelKey="uddi:uddi.org:categorization:types"/>
       <keyedReference keyName="uddi-org:types:checked"
          keyValue="checked"
          tModelKey="uddi:uddi.org:categorization:types"/>
      </categoryBag>
</tModel>
```

11.1.9.3 Valid Values

The valid values of this categorization system are the following strings:

- "entityKey"
- "businessKey"
- "tModelKey"
- "serviceKey"
- "bindingKey"
- "subscriptionKey"

11.1.9.4 Example of Use

The V3 version of the owningBusiness Category System has the following extra keyedReference added to its tModel to declare that the valid values of the owningBusiness value set are business keys:

```
<keyedReference keyName="entityKeyValues"
      keyValue="businessKey"
      tModelKey="uddi:uddi.org:categorization:entitykeyvalues"/>
```

11.2 UDDI Registry API tModels

UDDI defines a number of tModels to represent the UDDI application programming interface. Each of the core tModels are listed in this section. Every registry is required to register these tModels whether Web services that conform to these specifications are offered or not. Equivalent registry tModels from prior versions can be found in prior versions of the UDDI specification.

11.2.1 UDDI Inquiry API

11.2.1.1 Introduction

The group of APIs represented by this tModel deals with finding and retrieving information from the registry. To be a UDDI Version 3 compliant registry at least one of the nodes in a registry must provide a Web service that implements this tModel. See Section 5.1 *Inquiry API Set.*

11.2.1.2 Design Goals

The UDDI inquiry API provides a simple, complete set of programming interfaces that are used to:

- Search a UDDI registry to locate registry entries meeting a particular technical or business need.
- Retrieve details of registry entries once they have been found.

11.2.1.3 tModel Definition

Name:	uddi-org:inquiry_v3
Description:	UDDI Inquiry API Version 3 - Core Specification
UDDI Key (V3):	uddi:uddi.org:v3_inquiry
Derived V1,V2 format key:	uuid:01b9bbff-a8f5-3735-9a5e-5ea5ade7daaf
Categorization:	specification, xmlSpec, soapSpec, wsdlSpec

11.2.1.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:v3_inquiry">
  <name>uddi-org:inquiry_v3</name>
  <description>UDDI Inquiry API V3.0</description>
  <overviewDoc>
    <overviewURL useType="wsdlInterface">
       http://uddi.org/wsdl/uddi_api_v3_binding.wsdl#UDDI_Inquiry_SoapBinding
    </overviewURL>
  </overviewDoc>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#InqV3
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:wsdl"
      keyValue="wsdlSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:soap"
      keyValue="soapSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:xml"
      keyValue="xmlSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:specification"
      keyValue="specification"
      tModelKey="uddi:uddi.org:categorization:types"/>
   </categoryBag>
</tModel>
```

11.2.1.4 Programming Interfaces Covered

The UDDI APIs covered by this tModel are:

- find_binding: Used to locate specific bindings within a registered businessService. Returns a bindingDetail structure.
- find_business: Used to locate information about one or more businesses. Returns a businessList structure.
- find_relatedBusinesses: Used to locate information about businessEntity structures that are related to a specific business entity whose key is passed in the inquiry. The relatedBusinesses feature was introduced in UDDI version 2 and is used to manage registration of business units and subsequently relate them based on organizational hierarchies or business partner relationships. Returns a relatedBusinesses structure.
- find_service: Used to locate specific businessService structures within a registered businessEntity. Returns a serviceList structure.
- find_tModel: Used to locate one or more tModel information structures. Returns a tModelList structure.
- get_bindingDetail: Used to get full bindingTemplate information suitable for making one or more service requests. Returns a bindingDetail structure.
- get_businessDetail: Used to get the full businessEntity information for one or more businesses or organizations. Returns a businessDetail structure.
- get_businessDetailExt: Used to get extended businessEntity information. Returns a businessDetailExt structure.
- get_operationalInfo: Used to obtain metadata associated with core entities. Returns an operationalInfo structure.
- get_serviceDetail: Used to get full details for a given set of registered businessService date. Returns a serviceDetail structure.
- get_tModelDetail: Used to get full details for a given set of registered tModel data. Returns a tModelDetail structure.

11.2.1.5 Example of Use

The following is a typical bindingTemplate for a Web Service that implements the UDDI Version 3 Inquiry API:

```
<bindingTemplate bindingKey="…" serviceKey="…">
  <description>UDDI Inquiry API V3</description>
  <accessPoint useType="endpoint">
     http://URL_of_service
  </accessPoint>
  <tModelInstanceDetails>
    <tModelInstanceInfo
      tModelKey="uddi:uddi.org:v3_inquiry">
      <instanceDetails>
        <instanceParms>
          <![CDATA[
            <?xml version="1.0" encoding="utf-8" ?>
              <UDDIinstanceParmsContainer
               xmlns="urn:uddi-org:policy_v3_instanceParms">
                <defaultSortOrder>
                  uddi:uddi.org:sortorder:binarysort
                </defaultSortOrder>
              </UDDIinstanceParmsContainer>
          ]]>
        </instanceParms>
```

```
      </instanceDetails>
    </tModelInstanceInfo>
  </tModelInstanceDetails>
</bindingTemplate>
```

11.2.2 UDDI Publication API

11.2.2.1 Introduction

The group of programming interfaces represented by this tModel deals with adding or modifying information in the registry. While a UDDI Version 3 compliant registry is not required to have any of its nodes provide a Web service that implements this tModel, most do implement this tModel as a standard mechanism for providing the data on which the UDDI Inquiry API is based. See Section 5.2 *Publication API Set*.

11.2.2.2 Design Goals

The UDDI publication API provides a simple, complete set of programming interfaces that publishers of registry entries can use to publish UDDI registry entries.

11.2.2.3 tModel Definition

Name:	uddi-org:publication_v3
Description:	UDDI Publication API Version 3
UDDI Key (V3):	uddi:uddi.org:v3_publication
Derived V1,V2 format key:	uuid:72ade754-c6cc-315b-b014-7c94791fe15c
Categorization:	specification, xmlSpec, soapSpec, wsdlSpec

11.2.2.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:v3_publication">
  <name>uddi-org:publication_v3</name>
  <description>UDDI Publication API V3.0</description>
  <overviewDoc>
    <overviewURL useType="wsdlInterface">
       http://uddi.org/wsdl/uddi_api_v3_binding.wsdl#UDDI_Publication_SoapBinding
    </overviewURL>
  </overviewDoc>
  <overviewDoc>
    <overviewURL useType="text">
       http://uddi.org/pubs/uddi_v3.htm#PubV3
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:wsdl"
      keyValue="wsdlSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:soap"
      keyValue="soapSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:xml"
      keyValue="xmlSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:specification"
      keyValue="specification"
      tModelKey="uddi:uddi.org:categorization:types"/>
```

```
    </categoryBag>
</tModel>
```

11.2.2.4 Programming Interfaces Covered

The UDDI APIs covered by this tModel are:

- add_publisherAssertions: Used to add relationship assertions to the existing set of assertions. See Appendix A *Relationships and Publisher Assertions* for more information.
- delete_binding: Used to remove an existing bindingTemplate from the bindingTemplates collection that is part of a specified businessService structure.
- delete_business: Used to delete registered businessEntity information from the registry.
- delete_publisherAssertions: Used to delete specific assertions from the assertion set managed by a particular publisher account. Only effects the relationship assertions specified, causing any relationships formed by virtue of a prior assertion to be invalidated.
- delete_service: Used to delete an existing businessService from the businessServices collection that is part of a specified businessEntity.
- delete_tModel: Used to hide registered information about a tModel. Any tModel hidden in this way is still usable for reference purposes, but is simply hidden from find_tModel result sets. There is no way to actually cause a tModel to be deleted, except by administrative petition.
- get_assertionStatusReport: Used to get a list of relationship assertions that is useful for display in tools that help an administrator manage active and tentative assertions regarding relationships. Relationships help manage complex business structures that require more than one businessEntity or more than one publisher account to manage parts of a businessEntity. Returns an assertionStatusReport that includes the status of all assertions made involving any businessEntity controlled by the requesting publisher account.
- get_publisherAssertions: Used to get a list of active relationship assertions that are controlled by an individual publisher account. Returns a publisherAssertions structure containing a document that contains all relationship assertions associated with a specific publisher account.
- get_registeredInfo: Used to request an abbreviated synopsis of all information currently managed by a given individual.
- save_binding: Used to register new bindingTemplate information or update existing bindingTemplate information. Use this to control information about technical capabilities exposed by a registered business.
- save_business: Used to register new businessEntity information or update existing businessEntity information. Use this to control the full set of information about the entire business. Of the save_xx API's this one has the broadest effect.
- save_service: Used to register or update complete information about a businessService exposed by a specified businessEntity.
- save_tModel: Used to register or update complete information about a tModel.
- set_publisherAssertions: Used to save the complete set of relationship assertions for an individual publisher account. Replaces any existing assertions, and causes any old assertions that are not reasserted to be removed from the registry.

11.2.2.5 Example of Use

The following is a typical bindingTemplate for a Web Service that implements the UDDI Version 3 Publication API:

```
<bindingTemplate bindingKey="…" serviceKey="…">
  <description>UDDI Publication API V3</description>
  <accessPoint useType="endpoint">
     https://URL_of_service
  </accessPoint>
  <tModelInstanceDetails>
    <tModelInstanceInfo
      tModelKey="uddi:uddi.org:v3_publication">
      <instanceDetails>
        <instanceParms>
          <![CDATA[
            <?xml version="1.0" encoding="utf-8" ?>
              <UDDIinstanceParmsContainer
               xmlns="urn:uddi-org:policy_v3_instanceParms">
                <authInfoUse>required</authInfoUse>
              </UDDIinstanceParmsContainer>
          ]]>
        </instanceParms>
      </instanceDetails>
    </tModelInstanceInfo>
    <tModelInstanceInfo
      tModelKey="uddi:uddi.org:protocol:serverauthenticatedssl3" />
  </tModelInstanceDetails>
</bindingTemplate>
```

11.2.3 UDDI Security API

11.2.3.1 Introduction

The group of programming interfaces represented by this tModel pertains to authenticating with a UDDI node. Each registry and its nodes define policies for registering, authenticating and authorizing publishers and these APIs offer one way to provide tokens for use in authorizing other API access..

11.2.3.2 Design Goals

The UDDI security API provides a simple, complete set of APIs that users of registry entries MAY use to obtain the security credentials necessary to use all or parts of UDDI registries that distinguish between publishers. See Section 5.3 *Security Policy API Set*.

11.2.3.3 tModel Definition

Name:	uddi-org:security_v3
Description:	UDDI Security API Version 3
UDDI Key (V3):	uddi:uddi.org:v3_security
Derived V1,V2 format key:	uuid:e4cd70e2-22ec-3032-b1e6-cc31a9d55935
Categorization:	specification, xmlSpec, soapSpec, wsdlSpec

11.2.3.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:v3_security">
  <name>uddi-org:security_v3</name>
  <description>UDDI Security API V3.0</description>
  <overviewDoc>
    <overviewURL useType="wsdlInterface">
       http://uddi.org/wsdl/uddi_api_v3_binding.wsdl#UDDI_Security_SoapBinding
    </overviewURL>
  </overviewDoc>
  <overviewDoc>
    <overviewURL useType="text">
       http://uddi.org/pubs/uddi_v3.htm#SecV3
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:wsdl"
      keyValue="wsdlSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:soap"
      keyValue="soapSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:xml"
      keyValue="xmlSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:specification"
      keyValue="specification"
      tModelKey="uddi:uddi.org:categorization:types"/>
   </categoryBag>
</tModel>
```

11.2.3.4 Programming Interfaces Covered

The UDDI APIs covered by this tModel are:

- discard_authToken: Used to inform a UDDI node that a previously obtained authentication token is no longer required and should be considered invalid if used after this call is received.
- get_authToken: Used to request an authentication token in the form of an authInfo element from a UDDI node. An authInfo element MAY be required when using the API calls defined in Section 5.1 *Inquiry API Set*, Section 5.2 *Publication API Set*, Section 5.4 *Custody and Ownership Transfer API Set,* and Section 5.5 *Subscription API Set*.

11.2.3.5 Example of Use

The following is a typical bindingTemplate for a Web Service that implements the UDDI Version 3 Security API:

```
<bindingTemplate bindingKey="…" serviceKey="…">
  <description>UDDI Security API V3</description>
  <accessPoint useType="endpoint">
     https://URL_of_service
  </accessPoint>
  <tModelInstanceDetails>
    <tModelInstanceInfo tModelKey="uddi:uddi.org:v3_security"/>
    <tModelInstanceInfo
      tModelKey="uddi:uddi.org:protocol:serverauthenticatedssl3"/>
  </tModelInstanceDetails>
</bindingTemplate>
```

11.2.4 UDDI Replication API

11.2.4.1 Introduction

The group of programming interfaces represented by this tModel deals with managing the replication of information between nodes in a UDDI registry. See Section 7.4 *Replication API Set.* Nodes in multi-node UDDI registries SHOULD each offer Web services that conform to this specification.

11.2.4.2 Design Goals

The UDDI Replication API set provides a simple, complete set of APIs that UDDI nodes can use to replicate custodial data with other nodes in a multi-node registry.

11.2.4.3 tModel Definition

Name:	uddi-org:replication_v3
Description:	UDDI Replication API Version 3
UDDI Key (V3):	uddi:uddi.org:v3_replication
Derived V1,V2 format key:	uuid:998053a9-8672-3bf3-908a-c82deb4baecf
Categorization:	specification, xmlSpec, soapSpec, wsdlSpec

:

11.2.4.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:v3_replication">
  <name>uddi-org:replication_v3</name>
  <description>UDDI Replication API V3.0</description>
  <overviewDoc>
    <overviewURL useType="wsdlInterface">
       http://uddi.org/wsdl/uddi_repl_v3_binding.wsdl
    </overviewURL>
  </overviewDoc>
  <overviewDoc>
    <overviewURL useType="text">
       http://uddi.org/pubs/uddi_v3.htm#Repl
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:wsdl"
      keyValue="wsdlSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:soap"
      keyValue="soapSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:xml"
      keyValue="xmlSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:specification"
      keyValue="specification"
      tModelKey="uddi:uddi.org:categorization:types"/>
   </categoryBag>
</tModel>
```

11.2.4.4 Programming Interfaces Covered

The UDDI APIs covered by this tModel are:

- do_ping: Used to discover the existence and replication readiness of a node.
- get_changeRecords: Used to initiate the replication of change records from one node to another.
- notify_changeRecordsAvailable: Used to notify other nodes that the sending node has change data to be replicated.
- get_highWaterMarks: Used to obtain the high water marks from a node.

11.2.4.5 Example of Use

The following is a typical bindingTemplate for a Web Service that implements the UDDI Version 3 Replication API:

```
<bindingTemplate bindingKey="…" serviceKey="…">
  <description>UDDI Replication API V3</description>
  <accessPoint useType="endpoint">
     https://URL_of_service
  </accessPoint>
  <tModelInstanceDetails>
    <tModelInstanceInfo
      tModelKey="uddi:uddi.org:v3_replication" />
    <tModelInstanceInfo
      tModelKey="uddi:uddi.org:protocol:mutualauthenticatedssl3" />
  </tModelInstanceDetails>
</bindingTemplate>
```

11.2.5 UDDI Custody and Ownership Transfer API

11.2.5.1 Introduction

The group of programming interfaces represented by this tModel enables two publishers to cooperatively transfer ownership of one or more existing businessEntity or tModel structures from one publisher to another. See Section 5.4 *Custody and Ownership Transfer API Set.* Registries that offer a publishing Web service (uddi-org:v3_publish) SHOULD also offer a custody transfer Web service.

11.2.5.2 Design Goals

The UDDI Custody and Ownership Transfer API provides a simple, complete set of APIs that publishers can use to transfer ownership of UDDI entities from one publisher account to another and initiate custody transfer from one node to another.

11.2.5.3 tModel Definition

Name:	uddi-org:ownership_transfer_v3
Description:	UDDI Custody and Ownership Transfer API Version 3
UDDI Key (V3):	uddi:uddi.org:v3_ownership_transfer
Derived V1,V2 format key:	uuid:07ae0f8f-1bdc-32a7-b8dc-fe1d93d929a7
Categorization:	specification, xmlSpec, soapSpec, wsdlSpec

:

11.2.5.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:v3_ownership_transfer">
  <name>uddi-org:ownership_transfer_v3</name>
  <description>UDDI Custody and Ownership Transfer API V3.0</description>
  <overviewDoc>
    <overviewURL useType="wsdlInterface">
       http://uddi.org/wsdl/uddi_custody_v3_binding.wsdl
    </overviewURL>
  </overviewDoc>
  <overviewDoc>
```

```
    <overviewURL useType="text">
       http://uddi.org/pubs/uddi_v3.htm#OwnershipTransfer
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:wsdl"
      keyValue="wsdlSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:soap"
      keyValue="soapSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:xml"
      keyValue="xmlSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:specification"
      keyValue="specification"
      tModelKey="uddi:uddi.org:categorization:types"/>
   </categoryBag>
</tModel>
```

11.2.5.4 Programming Interfaces Covered

The UDDI APIs used to transfer ownership of entities and initiate custody transfer by this tModel are:

- get_transferToken: Used by a custodial publisher to initiate an ownership and/or custody transfer process at the custodial UDDI node.
- transfer_entities: Used by the target publisher to perform the transfer of ownership and/or initiate a custody transfer at the target node.

11.2.5.5 Example of Use

The following is a typical bindingTemplate for a Web Service that implements the UDDI Version 3 Custody and Ownership Transfer API set:

```
<bindingTemplate bindingKey="" serviceKey="…">
  <description>UDDI Custody and Ownership Transfer API V3</description>
  <accessPoint useType="endpoint">
     https://URL_of_service
  </accessPoint>
  <tModelInstanceDetails>
    <tModelInstanceInfo
      tModelKey="uddi:uddi.org:v3_ownership_transfer">
      <instanceDetails>
        <instanceParms>
          <![CDATA[
            <?xml version="1.0" encoding="utf-8" ?>
              <UDDIinstanceParmsContainer
               xmlns="urn:uddi-org:policy_v3_instanceParms">
                <authInfoUse>required</authInfoUse>
              </UDDIinstanceParmsContainer>
          ]]>
        </instanceParms>
      </instanceDetails>
    </tModelInstanceInfo>
    <tModelInstanceInfo
      tModelKey="uddi:uddi.org:protocol:serverauthenticatedssl3" />
  </tModelInstanceDetails>
</bindingTemplate>
```

11.2.6 UDDI Node Custody Transfer API

11.2.6.1 Introduction

The programming interface represented by this tModel enables two nodes in a registry to cooperatively transfer custody of one or more existing businessEntity or tModel structures from one node to another and at the same time to transfer ownership of the entities from one publisher to another. The API represented by this tModel is used to complete an inter-node custody transfer. See Section 5.4 *Custody and Ownership Transfer API Set.* Multi-node registries that offer a publishing Web service (uddi-org:v3_publish) and a custody transfer Web service (uddi-org:v3_custody) SHOULD offer the node custody transfer API.

11.2.6.2 Design Goals

The UDDI Node Custody Transfer API provides an API that UDDI nodes can use to complete a publisher request to transfer custody of UDDI entities from one node to another.

11.2.6.3 tModel Definition

Name:	uddi-org:node_custody_transfer_v3
Description:	UDDI Node Custody Transfer API Version 3
UDDI Key (V3):	uddi:uddi.org:v3_node_custody_transfer
Derived V1,V2 format key:	uuid:215c7844-5e81-347c-a2bf-54023ad463c8
Categorization:	specification, xmlSpec, soapSpec, wsdlSpec

11.2.6.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:v3_node_custody_transfer">
  <name>uddi-org:node_custody_transfer_v3</name>
  <description>UDDI Node Custody Transfer API V3.0</description>
  <overviewDoc>
    <overviewURL useType="wsdlInterface">
       http://uddi.org/wsdl/uddi_custody_v3_binding.wsdl
    </overviewURL>
  </overviewDoc>
  <overviewDoc>
    <overviewURL useType="text">
       http://uddi.org/pubs/uddi_v3.htm#NodeCustodyTransfer
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:wsdl"
      keyValue="wsdlSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:soap"
      keyValue="soapSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:xml"
      keyValue="xmlSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:specification"
      keyValue="specification"
      tModelKey="uddi:uddi.org:categorization:types"/>
   </categoryBag>
</tModel>
```

11.2.6.4 Programming Interfaces Covered

The UDDI API covered by this tModel is:

- transfer_custody: In a multi-node custody transfer, used when processing transfer_entities by the target node to verify acceptability of the transfer with the custodial node and to initiate the process of transferring the data between nodes.

11.2.6.5 Example of Use

The following is a typical bindingTemplate for a Web Service that implements the UDDI Version 3 Custody Transfer API:

```
<bindingTemplate bindingKey="…" serviceKey="…">
  <description>UDDI Node Custody Transfer API V3</description>
  <accessPoint useType="endpoint">
     https://URL_of_service
  </accessPoint>
  <tModelInstanceDetails>
    <tModelInstanceInfo
      tModelKey="uddi:uddi.org:v3_node_custody_transfer">
    </tModelInstanceInfo>
    <tModelInstanceInfo
      tModelKey="uddi:uddi.org:protocol:mutualauthenticatedssl3"/>
  </tModelInstanceDetails>
</bindingTemplate>
```

11.2.7 UDDI Value Set Caching API

11.2.7.1 Introduction

The programming interface represented by this tModel deals with obtaining a set of values for a checked value set so UDDI nodes may perform validation of publisher references themselves using the cached values obtained from such a Web service. See Section 5.6 *Value Set API Set.* Providers of registry checked value sets SHOULD offer a Web service that conforms to this specification.

11.2.7.2 Design Goals

The version 3.0 UDDI Value Set API defines programming interfaces that may be used by UDDI registries in the validation of references to checked category and identifier systems. The Value Set Caching API is a subset of the Value Set API. The usage restrictions can vary from simple to complex. A common example is insisting that the keyValues in keyedReferences come from a well-defined set. The UDDI registry and value set provider may agree that the UDDI nodes may cache the set of valid values. The provider may offer, and UDDI nodes use, a Web service to obtain the entire set of valid values at one time.

11.2.7.3 tModel Definition

Name:	uddi-org:valueSetCaching_v3
Description:	UDDI Value Set Caching API Version 3
UDDI Key (V3):	uddi:uddi.org:v3_valuesetcaching
Derived V1,V2 format key:	uuid:a24d9150-cdbb-3cb4-8843-41a5d0547170
Categorization:	specification, xmlSpec, soapSpec, wsdlSpec

11.2.7.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:v3_valuesetcaching">
  <name>uddi-org:valueSetCaching_v3</name>
  <description>UDDI Value Set Caching API V3.0</description>
  <overviewDoc>
    <overviewURL useType="wsdlInterface">
       http://uddi.org/wsdl/uddi_vscache_v3_binding.wsdl
    </overviewURL>
  </overviewDoc>
  <overviewDoc>
    <overviewURL useType="text">
       http://uddi.org/pubs/uddi_v3.htm#VSCaching
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:wsdl"
      keyValue="wsdlSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:soap"
      keyValue="soapSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:xml"
      keyValue="xmlSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:specification"
      keyValue="specification"
      tModelKey="uddi:uddi.org:categorization:types"/>
   </categoryBag>
</tModel>
```

11.2.7.4 Programming Interface Covered

The UDDI version 3 Value Set Caching API consists of one optional API:

- get_allValidValues: A UDDI registry that supports caching of external value sets may send the get_allValidValues API to the appropriate external Web service to refresh a cache of valid values for a category or identifier system that has granted permission to have its values cached.

11.2.7.5 Example of Use

The following is a typical bindingTemplate for a Web Service that implements the UDDI Version 3 Value Set Caching API. This Web service is registered by a value set provider:

```
<bindingTemplate bindingKey="" serviceKey="…">
  <description>UDDI Value Set Caching API V3</description>
  <accessPoint useType="endpoint">
     http://URL_of_service
  </accessPoint>
  <tModelInstanceDetails>
    <tModelInstanceInfo
      tModelKey="uddi:uddi.org:v3_valuesetcaching" />
  </tModelInstanceDetails>
</bindingTemplate>
```

11.2.8 UDDI Value Set Validation API

11.2.8.1 Introduction

The programming interface represented by this tModel deals with validating publisher use of category and identifier systems when describing their businessEntity, businessService, and tModel structures. See Section 5.6 *Value Set API Set.* Providers of externally checked value sets SHOULD offer a Web service that conforms to this specification.

11.2.8.2 Design Goals

The version 3.0 UDDI Value Set API defines programming interfaces that may be used by UDDI registries in the validation of references to checked value sets. The Value Set Validation API is a subset of the Value Set API and consists of a single programming interface that may be offered by a value set provider and used by UDDI nodes to validate references to checked value sets. The usage restrictions can vary from simple to complex.

11.2.8.3 tModel Definition

Name:	uddi-org:valueSetValidation_v3
Description:	UDDI Value Set Validation API Version 3
UDDI Key (V3):	uddi:uddi.org:v3_valuesetvalidation
Derived V1,V2 format key:	uuid:056fc4a2-bea3-30e5-8382-6d61e1ee23ce
Categorization:	specification, xmlSpec, soapSpec, wsdlSpec

11.2.8.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:v3_valuesetvalidation">
  <name>uddi-org:valueSetValidation_v3</name>
  <description>UDDI Value Set Validation API V3.0</description>
  <overviewDoc>
    <overviewURL useType="wsdlInterface">
       http://uddi.org/wsdl/uddi_vs_v3_binding.wsdl
    </overviewURL>
  </overviewDoc>
  <overviewDoc>
    <overviewURL useType="text">
       http://uddi.org/pubs/uddi_v3.htm#VSValid
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:wsdl"
      keyValue="wsdlSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:soap"
      keyValue="soapSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:xml"
      keyValue="xmlSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:specification"
      keyValue="specification"
      tModelKey="uddi:uddi.org:categorization:types"/>
   </categoryBag>
</tModel>
```

11.2.8.4 Programming Interface Covered

The UDDI version 3 Value Set Validation API consists of one optional API:

- validate_values: A UDDI registry that supports externally validated value sets may send the validate_values API to the appropriate external validation Web service when a publisher saves data that uses a value set whose use is regulated by the external party who controls that Web service. The normal use is to verify that specific categories or identifiers (keyValues) exist within the given value set. For certain value sets, the party providing the validation Web service may further restrict the use of a value based on other information known about the publisher or passed in the API.

11.2.8.5 Example of Use

The complete process a value set provider must follow to publish and activate a validation Web service for checked value set is determined by the registry's policy as detailed in Section 9.6.5 *Value Set Policies*. The following is a typical bindingTemplate for a Web Service that implements the UDDI Version 3 Value Set Validation API. This Web service is registered by a value set provider:

```
<bindingTemplate bindingKey="" serviceKey="…">
  <description>UDDI Value Set Validation API V3</description>
  <accessPoint useType="endpoint">
     http://URL_of_service
  </accessPoint>
  <tModelInstanceDetails>
    <tModelInstanceInfo
      tModelKey="uddi:uddi.org:v3_valuesetvalidation" />
  </tModelInstanceDetails>
</bindingTemplate>
```

11.2.9 UDDI Subscription API

11.2.9.1 Introduction

The group of programming interfaces represented by this tModel deals with the subscription of changed data from a UDDI registry. UDDI nodes can implement a subscription capability based on this API if they choose, but are not required to do so. The UDDI Subscription API is comprised of two complementary Web services: one that is implemented by a registry to enable and manage subscriptions and notify subscribers of changes; and one that can be implemented by subscribers to receive update notifications from a UDDI registry. See Section 5.5 *Subscription API Set.*

11.2.9.2 Design Goals

The UDDI Subscription API provides a simple, complete set of programming interfaces that subscribers of registry entries can use and/or implement to obtain updates to a UDDI registry. This API is comprised of two parts. The subscription capability is covered here. The subscriber portion of the API is covered in Section 11.2.10 *UDDI Subscription Listener API*.

11.2.9.3 tModel Definition

Name:	uddi-org:subscription_v3
Description:	UDDI Subscription API
UDDI Key (V3):	uddi:uddi.org:v3_subscription
Derived V1,V2 format key:	uuid:c6eb3d94-8051-3fbb-9320-a6147e266e57

Categorization: specification, xmlSpec, soapSpec, wsdlSpec

11.2.9.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:v3_subscription">
  <name>uddi-org:subscription_v3</name>
  <description>UDDI Subscription API V3.0</description>
  <overviewDoc>
    <overviewURL useType="wsdlInterface">
       http://uddi.org/wsdl/uddi_sub_v3_binding.wsdl
    </overviewURL>
  </overviewDoc>
  <overviewDoc>
    <overviewURL useType="text">
       http://uddi.org/pubs/uddi_v3.htm#Sub
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:wsdl"
      keyValue="wsdlSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:soap"
      keyValue="soapSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:xml"
      keyValue="xmlSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:specification"
      keyValue="specification"
      tModelKey="uddi:uddi.org:categorization:types"/>
   </categoryBag>
</tModel>
```

11.2.9.4 Programming Interfaces Covered

The UDDI APIs that are covered by this tModel are:

- delete_subscription: Used to remove an existing subscription from the UDDI node. Notifications of changes will cease.
- get_subscriptions: Used by a publisher to obtain all of the subscriptions that are in effect for that publisher's account.
- get_notification: Used to request that a notification be resent.
- save_subscription: Used to register new subscription information or update existing subscription information.

11.2.9.5 Example of Use

The following is a typical bindingTemplate for a Web Service that implements the UDDI Version 3 Subscription API:

```
<bindingTemplate bindingKey="…" serviceKey="…">
  <description>UDDI Subscription API V3</description>
  <accessPoint useType="endpoint">
     https://URL_of_service
  </accessPoint>
  <tModelInstanceDetails>
    <tModelInstanceInfo
      tModelKey="uddi:uddi.org:v3_subscription">
      <instanceDetails>
        <instanceParms>
          <![CDATA[
            <?xml version="1.0" encoding="utf-8" ?>
              <UDDIinstanceParmsContainer
               xmlns="urn:uddi-org:policy_v3_instanceParms">
                <authInfoUse>required</authInfoUse>
                <filterUsingFindAPI>supported</filterUsingFindAPI>
              </UDDIinstanceParmsContainer>
          ]]>
        </instanceParms>
      </instanceDetails>
    </tModelInstanceInfo>
    <tModelInstanceInfo
      tModelKey="uddi:uddi.org:protocol:serverauthenticatedssl3" />
  </tModelInstanceDetails>
</bindingTemplate>
```

11.2.10 UDDI Subscription Listener API

11.2.10.1 Introduction

This tModel complements the UDDI Subscription API. The programming interface represented by this tModel deals with receiving notifications from a UDDI node that implements the UDDI Subscription API. This API is implemented by UDDI subscribers when they wish to receive asynchronous notification of changes to UDDI entities. See Section 5.5 *Subscription API Set.*

11.2.10.2 Design Goals

The version 3.0 UDDI SubscriptionListener API defines a programming interface to receive asynchronous information about changes to UDDI entities that the subscriber previously expressed interest in using the UDDI Subscription Listener API.

11.2.10.3 tModel Definition

Name:	uddi-org:subscriptionListener_v3
Description:	UDDI Subscription Listener API
UDDI Key (V3):	uddi:uddi.org:v3_subscriptionlistener
Derived V1,V2 format key:	uuid:0f965bee-b120-3a66-bdc2-4908819c1174
Categorization:	specification, xmlSpec, soapSpec, wsdlSpec

11.2.10.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:v3_subscriptionlistener">
  <name>uddi-org:subscriptionListener_v3</name>
  <description>UDDI Subscription Listener API V3.0</description>
  <overviewDoc>
    <overviewURL useType="wsdlInterface">
        http://uddi.org/wsdl/uddi_subr_v3_binding.wsdl
    </overviewURL>
  </overviewDoc>
  <overviewDoc>
    <overviewURL useType="text">
       http://uddi.org/pubs/uddi_v3.htm#Subscribe
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:wsdl"
      keyValue="wsdlSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:soap"
      keyValue="soapSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:xml"
      keyValue="xmlSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:specification"
      keyValue="specification"
      tModelKey="uddi:uddi.org:categorization:types"/>
   </categoryBag>
</tModel>
```

11.2.10.4 Programming Interfaces Covered

The UDDI version 3 Subscription Listener API consists of a single optional API:

- notify_subscriptionListener: A UDDI node that supports subscription uses this programming interface to provide programmatic notification of records matching the criteria of a saved subscription to subscribers that have indicated they should receive asynchronous updates to subscribed UDDI entities using SOAP.

11.2.10.5 Example of Use

The following is a typical bindingTemplate for a Web Service that implements the UDDI Version 3 Subscription Listener API. This Web service is registered by a subscriber of asynchronous updates to UDDI entities:

```
<bindingTemplate bindingKey="…" serviceKey="…">
  <description>UDDI Subscription Listener API V3</description>
  <accessPoint useType="endpoint">
     http://URL_of_service
  </accessPoint>
  <tModelInstanceDetails>
    <tModelInstanceInfo
      tModelKey="uddi:uddi.org:v3_subscriptionlistener" />
  </tModelInstanceDetails>
</bindingTemplate>
```

11.3 Transport and Protocol tModels

tModels of the transport and protocol sort are referenced by service bindings to describe the type of protocol or transport used to invoke the service. This is often done when other parts of the technical fingerprint (other tModels referenced in the same bindingTemplate) are silent or ambiguous with respect to the protocol or transport used by the particular service binding, or when the provider of the Web service wants to explicitly call out the protocol or transport in the technical fingerprint, to enable proper discovery. Protocol and transport tModels are frequently coupled with other tModels that describe what the service does in detail.

The tModels that follow correspond to protocols and transports that are recommended for different parts of the UDDI API as described in Section 9.6.2 *Information Model.* The bindingTemplate structures for the UDDI API described in Section 11.1.9 *UDDI Registry API tModels* SHOULD reference one or more of these tModels.

11.3.1 Secure Sockets Layer Version 3 with Server Authentication

11.3.1.1 Introduction

The use of Secure Sockets Layer Version 3.0 (SSL 3.0)[41] for transport confidentiality and server authentication as an application protocol is represented by this tModel. The server authentication on SSL 3.0 represented by this tModel involves the presentation of a server side certificate to the client that initiates the SSL 3.0 connection. Web services that require use of SSL 3.0 SHOULD reference this tModel in their bindingTemplate structures.

11.3.1.2 Design Goals

There are two principal design goals for the SSL 3.0 with server authentication tModel. The first is to separate the use of SSL 3.0 described from the application protocol HTTP and also from the higher level messaging protocol, such as SOAP. The second goal for this tModel is to facilitate discovery of Web services that support SSL 3.0 for transport confidentiality and server authentication.

11.3.1.3 tModel Definition

This tModel is used in bindingTemplate elements of Web services to indicate that the Web service implementation requires the use of SSL 3.0 with server authentication.

Name:	uddi-org:serverAuthenticatedSSL3
Description:	Secure Sockets Layer 3.0 with Server Authentication
UDDI Key (V3):	uddi:uddi.org:protocol:serverauthenticatedssl3
Derived V1,V2 format key:	uuid:c8aea832-3faf-33c6-b32a-bbfd1b926294
Categorization:	protocol

[41] http://www.netscape.com/eng/ssl3/index.html

11.3.1.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel
 tModelKey="uddi:uddi.org:protocol:serverauthenticatedssl3">
  <name>uddi-org:serverAuthenticatedSSL3</name>
  <description>Secure Sockets Layer Version 3.0 with Server
               Authentication</description>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#serverSSL3
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:protocol"
      keyValue="protocol"
      tModelKey="uddi:uddi.org:categorization:types"/>
   </categoryBag>
</tModel>
```

11.3.1.4 Example of Use

The following is an example of a bindingTemplate for a UDDI Publish API Web service that uses the SSLv3 with server authentication tModel to represent the transport layer:

```
<bindingTemplate bindingKey="uddi:...." serviceKey="uddi:....">
  <description>UDDI Publication API V3</description>
  <accessPoint useType="endpoint">
     https://URL_of_service
  </accessPoint>
  <tModelInstanceDetails>
    <tModelInstanceInfo
      tModelKey="uddi:uddi.org:v3_publication" />
    <tModelInstanceInfo
      tModelKey="uddi:uddi.org:protocol:serverauthenticatedssl3"/>
  </tModelInstanceDetails>
</bindingTemplate>
```

11.3.2 Secure Sockets Layer Version 3 with Mutual Authentication

11.3.2.1 Introduction

The use of Secure Sockets Layer Version 3.0 (SSL 3.0)[42] for transport confidentiality and mutual authentication as an application protocol is represented by this tModel. Mutual authentication as represented by this tModel includes both client and server authentication. The server authentication using SSL 3.0 represented by this tModel involves the presentation of a server side certificate to the client that initiates the SSL 3.0 connection. The client authentication using SSL 3.0 represented by this tModel involves the server issuing a certificate request for a client certificate and the client sending a client certificate as described in the SSL 3.0 specification. Web services that require SSL 3.0 Mutual Authentication SHOULD reference this tModel in their bindingTemplate structures.

[42] http://www.netscape.com/eng/ssl3/index.html

11.3.2.2 Design Goals

There are two principal design goals for the SSL 3.0 with mutual authentication tModel. The first is to separate the use of SSL 3.0 described from the application protocol HTTP and also from the higher level messaging protocol, such as SOAP. The second goal for this tModel is to facilitate discovery of UDDI Web services that support SSL 3.0 for transport confidentiality and mutual authentication.

11.3.2.3 tModel Definition

This tModel is used in bindingTemplate elements of Web services to indicate that the Web service implementation requires the use of SSL 3.0 with mutual authentication.

Name:	uddi-org:mutualAuthenticatedSSL3
Description:	Secure Sockets Layer 3.0 with Mutual Authentication
UDDI Key (V3):	uddi:uddi.org:protocol:mutualauthenticatedssl3
Derived V1,V2 format key:	uuid:9555b5b6-55d4-3b0e-bb17-e084fed4e33f
Categorization:	protocol

11.3.2.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel
 tModelKey="uddi:uddi.org:protocol:mutualauthenticatedssl3">
  <name>uddi-org:mutualAuthenticatedSSL3</name>
  <description>Secure Sockets Layer Version 3.0 with Mutual
              Authentication</description>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#mutualSSL3
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:protocol"
      keyValue="protocol"
      tModelKey="uddi:uddi.org:categorization:types"/>
   </categoryBag>
</tModel>
```

11.3.2.4 Example of Use

The following is an example of a bindingTemplate for a UDDI Replication API Web service using the SSLv3 with mutual authentication tModel to represent the transport layer:

```
<bindingTemplate bindingKey="...." serviceKey="....">
  <description>UDDI Replication API V3</description>
  <accessPoint useType="endpoint">
     https://URL_of_service
  </accessPoint>
  <tModelInstanceDetails>
    <tModelInstanceInfo
      tModelKey="uddi:uddi.org:v3_replication" />
    <tModelInstanceInfo
      tModelKey=" uddi:uddi.org:protocol:mutualauthenticatedssl3" />
  </tModelInstanceDetails>
</bindingTemplate>
```

11.3.3 UDDI HTTP Transport

11.3.3.1 Introduction

In UDDI, tModels are used to establish the existence of a variety of concepts and to point to their technical definitions. Transport tModels are referenced by service bindings to describe the type of transport used to invoke the service, either when other parts of the technical fingerprint (other tModels referenced in the same bindingTemplate) do not specify the transport used by the particular service binding, or when the service provider wants to explicitly call out the transport in the technical fingerprint so as to enable discovery. Transport tModels are frequently coupled with other tModels that describe more completely what the service does. For example, the HTTP Transport tModel may be coupled with the SOAP Protocol tModel to indicate that a Web service communicates using SOAP over HTTP. The HTTP Transport tModel provides a means for designating that services transport messages using the HTTP protocol.

11.3.3.2 Design Goals

The HTTP Transport tModel is provided to enable discovery of services that transport messages using the HTTP protocol. This tModel can be used alone to provide a technical fingerprint for simple services that are accessed through HTTP when they might otherwise not have a tModel to reference in their bindingTemplates, or it can be used in conjunction with other protocol tModels to indicate the applicable transport when other parts of the technical fingerprint are ambiguous or silent with respect to the transport to use.

11.3.3.3 tModel Definition

This tModel is used to describe a Web service that transports messages using the HTTP protocol.

Name:	uddi-org:http
Description:	A Web service that uses HTTP transport
UDDI Key (V3):	uddi:uddi.org:transport:http
Evolved V1,V2 format key:	uuid:68DE9E80-AD09-469D-8A37-088422BFBC36
Categorization:	transport

:

11.3.3.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel
 tModelKey="uddi:uddi.org:transport:http">
  <name>uddi-org:http</name>
  <description> A Web service that uses HTTP transport</description>
  <overviewDoc>
    <overviewURL useType="text">
      http://uddi.org/pubs/uddi_v3.htm#overHTTP
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:transport"
      keyValue="transport"
      tModelKey="uddi:uddi.org:categorization:types"/>
```

```
   </categoryBag>
</tModel>
```

11.3.3.4 Example of Use

The following is a typical bindingTemplate for a simple service that references the HTTP Transport tModel:

```
<bindingTemplate bindingKey="uddi:...." serviceKey="uddi:....">
  <description xml:lang="en">Buy from Bigfoot Breweries over
    the Web</description>
  <accessPoint useType="endpoint"
    http://www.bigfootbreweries.example/shop.html
  </accessPoint>
  <tModelInstanceDetails>
    <tModelInstanceInfo
      tModelKey="uddi:uddi.org:transport:http />
  </tModelInstanceDetails>
</bindingTemplate>
```

The next example shows a bindingTemplate for a service whose technical fingerprint (the first tModel referenced) specifies two alternative access techniques, of which HTTP is one:

```
<bindingTemplate bindingKey="uddi:...." serviceKey="uddi:...">
  <description xml:lang="en">Obtain financing</description>
  <accessPoint useType="endpoint">
    http://www.consolidatedholdings.example/finance.html
  </accessPoint>
  <!-- The first tModel describes a technical interface (to
       obtain financing) which includes multiple binding
       descriptions.  HTTP is one such supported binding. -->
  <tModelInstanceDetails>
    <tModelInstanceInfo tModelKey="uddi:some.tmodelkey…"/>
  </tModelInstanceDetails>
  <!-- The second tModel indicates that this service instance
       is accessed using http -->
  <tModelInstanceDetails>
    <tModelInstanceInfo
      tModelKey="uddi:uddi.org:transport:http"/>
  </tModelInstanceDetails>
</bindingTemplate>
```

11.3.4 UDDI SMTP Transport

11.3.4.1 Introduction

In UDDI, tModels are used to establish the existence of a variety of concepts and to point to their technical definitions. tModels of the transport sort are referenced by service bindings to describe the type of transport used to invoke the service, either when other parts of the technical fingerprint (other tModels referenced in the same bindingTemplate) are silent or ambiguous with respect to the transport used by the particular service binding, or when the service provider wants to explicitly call out the transport in the technical fingerprint so as to enable proper discovery. Transport-type tModels are frequently coupled with other tModels that describe more completely what the service does. For very simple services, a transport tModel may be the only tModel referenced in a bindingTemplate. The SMTP transport tModel provides a means for designating those services that are invoked by sending an eMail to the accessPoint in the bindingTemplate.

11.3.4.2 Design Goals

The SMTP Transport tModel is provided to enable discovery of services that are invoked by sending an eMail to the address in the accessPoint element of the bindingTemplate, to mix-in the applicable transport when other parts of the technical fingerprint are ambiguous or silent with respect to the transport to use, and to enable simple services that are accessed through eMail to have a technical fingerprint when they might otherwise not have a tModel to reference in their bindingTemplates.

11.3.4.3 tModel Definition

This tModel is used to describe a Web service that is invoked through SMTP eMail transmissions. These transmissions may be either between people or applications.

Name:	uddi-org:smtp
Description:	E-mail based Web service
UDDI Key (V3):	uddi:uddi.org:transport:smtp
Evolved V1,V2 format key:	uuid:93335D49-3EFB-48A0-ACEA-EA102B60DDC6
Categorization:	transport

11.3.4.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel
 tModelKey="uddi:uddi.org:transport:smtp">
  <name>uddi-org:smtp</name>
  <description>E-mail based Web service</description>
  <overviewDoc>
    <overviewURL useType="text">
      http://uddi.org/pubs/uddi_v3.htm#overSMTP
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:transport"
      keyValue="transport"
      tModelKey="uddi:uddi.org:categorization:types"/>
   </categoryBag>
</tModel>
```

11.3.4.4 Example of Use

The following is a typical bindingTemplate that references the SMTP Transport tModel:

```
<bindingTemplate bindingKey="...." serviceKey="uddi:...">
  <description xml:lang="en">Send eMail to buy from
       Island Trading</description>
  <accessPoint useType="endpoint">
    mailto:order@islandtrading.example
  </accessPoint>
  <tModelInstanceDetails>
    <tModelInstanceInfo
      tModelKey="uddi:uddi.org:transport:smtp"/>
  </tModelInstanceDetails>
</bindingTemplate>
```

11.3.5 UDDI FTP Transport

11.3.5.1 Introduction

In UDDI, tModels are used to establish the existence of a variety of concepts and to point to their technical definitions. tModels of the transport sort are referenced by service bindings to describe the type of transport used to invoke the service, either when other parts of the technical fingerprint (other tModels referenced in the same bindingTemplate) are silent or ambiguous with respect to the transport used by the particular service binding, or when the service provider wants to explicitly call out the transport in the technical fingerprint so as to enable proper discovery. Transport-type tModels are frequently coupled with other tModels that describe more completely what the service does. For very simple services, a transport tModel may be the only tModel referenced in a bindingTemplate. The FTP transport tModel provides a means for designating those services that are invoked using the File Transfer Protocol on the accessPoint in the bindingTemplate.

11.3.5.2 Design Goals

The FTP Transport tModel is provided to enable discovery of services that are invoked by performing an FTP on the address in the accessPoint element of the bindingTemplate, to mix-in the applicable transport when other parts of the technical fingerprint are ambiguous or silent with respect to the transport to use, and to enable simple services that are accessed using FTP to have a technical fingerprint when they might otherwise not have a tModel to reference in their bindingTemplates.

11.3.5.3 tModel Definition

This tModel is used to describe a Web service that is invoked through file transfers via the FTP.

Name:	uddi-org:ftp
Description:	File Transfer Protocol (FTP) based Web service
UDDI Key (V3):	uddi:uddi.org:transport:ftp
Evolved V1,V2 format key:	uuid:5FCF5CD0-629A-4C50-8B16-F94E9CF2A674
Categorization:	transport

11.3.5.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel
 tModelKey="uddi:uddi.org:transport:ftp">
  <name>uddi-org:ftp</name>
  <description>File Transfer Protocol (FTP) based Web service</description>
  <overviewDoc>
    <overviewURL useType="text">
      http://uddi.org/pubs/uddi_v3.htm#overFTP
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:transport"
      keyValue="transport"
      tModelKey="uddi:uddi.org:categorization:types"/>
   </categoryBag>
</tModel>
```

11.3.5.4 Example of Use

The following is a typical bindingTemplate that references the FTP Transport tModel:

```
<bindingTemplate bindingKey="...." serviceKey="uddi:...">
  <description xml:lang="en">Obtain Island Trading product
 catalog using FTP</description>
  <accessPoint useType="endpoint">
    ftp://islandtrading.example/catalog
  </accessPoint>
  <tModelInstanceDetails>
    <tModelInstanceInfo
      tModelKey="uddi:uddi.org:transport:ftp"/>
  </tModelInstanceDetails>
</bindingTemplate>
```

11.3.6 UDDI Fax Transport

11.3.6.1 Introduction

In UDDI, tModels are used to establish the existence of a variety of concepts and to point to their technical definitions. tModels of the transport sort are referenced by service bindings to describe the type of transport used to invoke the service, either when other parts of the technical fingerprint (other tModels referenced in the same bindingTemplate) are silent or ambiguous with respect to the transport used by the particular service binding, or when the service provider wants to explicitly call out the transport in the technical fingerprint so as to enable proper discovery. Transport-type tModels are frequently coupled with other tModels that describe more completely what the service does. For very simple services, a transport tModel may be the only tModel referenced in a bindingTemplate. The fax transport tModel provides a means for designating those services that are invoked using the fax machine on the accessPoint in the bindingTemplate.

11.3.6.2 Design Goals

The Fax Transport tModel is provided to enable discovery of services that are invoked by sending a fax to the address in the accessPoint element of the bindingTemplate, to mix-in the applicable transport when other parts of the technical fingerprint are ambiguous or silent with respect to the transport to use, and to enable simple services that are accessed by sending a fax to have a technical fingerprint when they might otherwise not have a tModel to reference in their bindingTemplates.

11.3.6.3 tModel Definition

This tModel is used to describe a Web service that is invoked through fax transmissions. These transmissions may be either between people or applications.

Name:	uddi-org:fax
Description:	Fax-based Web service
UDDI Key (V3):	uddi:uddi.org:transport:fax
Evolved V1,V2 format key:	uuid:1A2B00BE-6E2C-42F5-875B-56F32686E0E7
Categorization:	transport

11.3.6.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel
 tModelKey="uddi:uddi.org:transport:fax">
  <name>uddi-org:fax</name>
  <description>Fax-based Web service</description>
  <overviewDoc>
    <overviewURL useType="text">
      http://uddi.org/pubs/uddi_v3.htm#overFax
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:transport"
      keyValue="transport"
      tModelKey="uddi:uddi.org:categorization:types"/>
   </categoryBag>
</tModel>
```

11.3.6.4 Example of Use

The following is a typical bindingTemplate that references the Fax Transport tModel. Note that the accessPoint contains a URI using the fax scheme[43].

```
<bindingTemplate bindingKey="...." serviceKey="uddi:…">
  <description xml:lang="en">Send facsimile to buy from
     Island Trading</description>
  <accessPoint useType="endpoint">fax:+1-800-555-5555</accessPoint>
  <tModelInstanceDetails>
    <tModelInstanceInfo tModelKey="uddi:uddi.org:transport:fax"/>
  </tModelInstanceDetails>
</bindingTemplate>
```

11.3.7 UDDI Telephone Transport

11.3.7.1 Introduction

In UDDI, tModels are used to establish the existence of a variety of concepts and to point to their technical definitions. tModels of the transport sort are referenced by service bindings to describe the type of transport used to invoke the service, either when other parts of the technical fingerprint (other tModels referenced in the same bindingTemplate) are silent or

[43] http://www.ietf.org/rfc/rfc2806

ambiguous with respect to the transport used by the particular service binding, or when the service provider wants to explicitly call out the transport in the technical fingerprint so as to enable proper discovery. Transport-type tModels are frequently coupled with other tModels that describe more completely what the service does. For very simple services, a transport tModel may be the only tModel referenced in a bindingTemplate. The telephone transport tModel provides a means for designating those services that are invoked using a voice over a telephone on the accessPoint in the bindingTemplate.

11.3.7.2 Design Goals

The Telephone Transport tModel is provided to enable discovery of services that are accessed using voice with a telephone, to mix-in the voice over telephone access technique when other parts of the technical fingerprint are ambiguous or silent with respect to the way the service is accessed, and to enable simple services that are accessed by using a telephone to have a technical fingerprint when they might otherwise not have a tModel to reference in their bindingTemplates.

11.3.7.3 tModel Definition

This tModel is used to describe a service that is invoked through a telephone call and interaction by voice and/or touch-tone.

Name:	uddi-org:telephone
Description:	Telephone based service
UDDI Key (V3):	uddi:uddi.org:transport:telephone
Evolved V1,V2 format key:	uuid:38E12427-5536-4260-A6F9-B5B530E63A07
Categorization:	transport

11.3.7.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel
 tModelKey="uddi:uddi.org:transport:telephone">
  <name>uddi-org:telephone</name>
  <description>Telephone based service</description>
  <overviewDoc>
    <overviewURL useType="text">
      http://uddi.org/pubs/uddi_v3.htm#overPhone
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:transport"
      keyValue="transport"
      tModelKey="uddi:uddi.org:categorization:types"/>
   </categoryBag>
</tModel>
```

11.3.7.4 Example of Use

The following is a typical bindingTemplate that references the Telephone Specification tModel. Note that the accessPoint contains a URI using the tel scheme[44].

```
<bindingTemplate bindingKey="...." serviceKey="uddi:...">
  <description xml:lang="en">Buy from Island Trading over
     the phone</description>
  <accessPoint useType="endpoint">tel:+1-800-555-5555</accessPoint>
  <tModelInstanceDetails>
    <tModelInstanceInfo
      tModelKey="uddi:uddi.org:transport:telephone"/>
  </tModelInstanceDetails>
</bindingTemplate>
```

11.4 Find Qualifier tModels

The tModels in this section are defined to be used when performing inquiry functions to qualify the inquiry criteria and describe treatment of the results. UDDI requires that all UDDI registries include these tModels. The designation "Support: Mandatory" indicates the tModel MUST be supported by UDDI nodes. Short names provided represent the short strings that can be used to identify the find qualifiers included herein. See Section 5.1.4 *Find Qualifiers* for more information.

In UDDI, tModels are used to establish the existence of a variety of concepts and to point to their technical definitions. tModels of the find qualifier sort are referenced by findQualifier elements in UDDI inquiries to describe behaviors related to the search input data and treatment of the result set.

11.4.1 UDDI SQL99 Approximate Match Find Qualifier

11.4.1.1 Introduction

The SQL 99 Approximate Match find qualifier allows wildcards to be specified in UDDI name and keyedReference elements according to the character version of the <like predicate> of section 8.5 of the SQL 99 specification[45].

11.4.1.2 Design Goals

The SQL99 Approximate Match tModel is provided to enable discovery of UDDI entities when exact matching criteria is unknown or too restrictive. Approximate matching is performed on UDDI name, keyValue, and where significant, keyName elements. Wildcards are denoted with a percent sign (%) to indicate any value for any number of characters and an underscore (_) to indicate any value for a single character. The backslash character (\) is used as an escape character for the percent sign, underscore and backslash characters.

11.4.1.3 tModel Definition

This tModel is a find qualifier that is used to enable approximate matching in UDDI inquiries. This tModel cannot be referenced in a find qualifier if the exactMatch find qualifier tModel is referenced. By default, case and diacritic sensitive matching is performed on any portion of the value provided.

---

[44] http://www.ietf.org/rfc/rfc2806

[45] SQL99, ISO/IEC9075-2:1999(E) section 8.5

Use of wildcards in the keyValue fields of keyedReferences takes precedence over the effects of category treatment find qualifiers such as andAllKeys, orAllKeys and orLikeKeys. This means that the individual elements from the set of keyValues that result from using a wildcard are not subject to the ANDing and ORing criteria described by these other find qualifiers, but the set itself is. For example, the inquiry of the form *find businesses with services in Georgia that relate to transporting goods*, is specified with *Georgia* as an explicit keyValue from a geographic value set, *transporting goods* as a partial keyValue from a product and services value set with a trailing wildcard, and the *andAllKeys* find qualifier (either explicitly specified, or omitted as it is the default behavior for the categoryBag). The inquiry is performed by finding the businesses that have the *Georgia* service location AND a product and services categorization starting with the product code for *transportation of goods*.

Name:	uddi-org:approximateMatch:SQL99
Short name:	approximateMatch
Description:	UDDI SQL99 approximate matching find qualifier
UDDI Key (V3):	uddi:uddi.org:findqualifier:approximatematch
Derived V1,V2 format key:	uuid:8af9e55a-5c35-30dd-915a-8a7961bc1054
Categorization:	findQualifier
Support:	Mandatory

:

11.4.1.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:findqualifier:approximatematch">
  <name>uddi-org:approximateMatch:SQL99</name>
  <description>UDDI approximate matching find qualifier
  </description>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#wildcard
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:findQualifier"
      keyValue="findQualifier"
      tModelKey="uddi:uddi.org:categorization:types"/>
  </categoryBag>
</tModel>
```

11.4.1.4 Example of Use

The following represent a typical inquiry that uses the SQL99 approximateMatch find qualifier. This example shows an inquiry using the SQL99 approximateMatch find qualifier to find all businesses that are either themselves categorized or that have businessService or bindingTemplate structures categorized with any of the categories in the UNSPSC family "Telephones and personal telecommunications devices and accessories". In this example a wildcard character is added to the end of the keyValue specified for a UNSPSC keyedReference because this value set is formatted hierarchically with more detailed nodes represented by additional numeric fragments on the right side of the value.

```
<find_business xmlns="urn:uddi-org:api_v3">
  <findQualifiers>
    <findQualifier>
       uddi:uddi.org:findqualifier:approximatematch
    </findQualifier>
    <findQualifier>
       uddi:uddi.org:findqualifier:combinecategorybags
    </findQualifier>
  </findQualifiers>
  <categoryBag>
    <keyedReference keyValue="34.10.%"
       tModelKey="uddi:uddi.org:ubr:categorization:unspsc"/>
  </categoryBag>
</find_business>
```

11.4.2 UDDI Exact Match Find Qualifier

11.4.2.1 Introduction

The Exact Match find qualifier represents the default behavior for matching name, keyValue, and keyName arguments used in UDDI inquiry functions. This qualifier directs inquiry functions to find matches based on exactly what has been provided in the input criteria of the inquiry.

11.4.2.2 Design Goals

The Exact Match find qualifier tModel is provided to explicitly request discovery of UDDI entities with an exact match on the name, keyValue, and where applicable, keyName arguments, after normalization.

11.4.2.3 tModel Definition

This tModel is a find qualifier that is used to enable exact matching in UDDI inquiries. When this tModel is referenced in a find qualifier, only entities with names, keyValues, and where applicable, keyNames that exactly match the values passed in the input criteria, after normalization, will be returned. This qualifier conflicts with any approximate match qualifier such as uddi-org:approximateMatch and with uddi-org:caseInsensitiveMatch.

Name:	uddi-org:exactMatch
Short name:	exactMatch
Description:	UDDI Exact Matching find qualifier
UDDI Key (V3):	uddi:uddi.org:findqualifier:exactmatch
Derived V1,V2 format key:	uuid:05a6e3ef-2e94-3c91-9360-a31cd335c758

Categorization: findQualifier

Support: Mandatory, default

11.4.2.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel
 tModelKey="uddi:uddi.org:findqualifier:exactmatch">
  <name>uddi-org:exactMatch</name>
  <description>UDDI exact name matching findQualifier
  </description>
  <overviewDoc>
    <overviewURL useType="text">
       http://uddi.org/pubs/uddi_v3.htm#exactmatch
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:findQualifier"
      keyValue="findQualifier"
      tModelKey="uddi:uddi.org:categorization:types"/>
  </categoryBag>
</tModel>
```

11.4.2.4 Example of Use

The following represents a typical inquiry that references the Exact Name Match find qualifier tModel. All businesses with the name 'My Company Name' are returned.

```
<find_business xmlns="urn:uddi-org:api_v3">
  <findQualifiers>
    <findQualifier>
       uddi:uddi.org:findqualifier:exactmatch
    </findQualifier>
  </findQualifiers>
  <name>My Company Name</name>
</find_business>
```

11.4.3 UDDI Case Insensitive Match Find Qualifier

11.4.3.1 Introduction

The Case Insensitive Match find qualifier allows matching to occur on any inquiry involving a name, keyValue, and where relevant, keyName, without regard to the case of the search argument(s).

11.4.3.2 Design Goals

The Case Insensitive Match find qualifier tModel is provided to enable discovery of UDDI entities when case sensitivity is not important.

11.4.3.3 tModel Definition

This tModel is a find qualifier that is used to enable case insensitive matching in UDDI inquiries. When this tModel is referenced in a find qualifier, case is irrelevant in the search results and all entries, independent of case, that match the value passed in the search arguments will be returned. This qualifier is ignored for languages that are not case-significant and cannot be used with the uddi-org:exactMatch or the uddi-org:caseSensitiveMatch find qualifiers.

Name:	uddi-org:caseInsensitiveMatch
Short name:	caseInsensitiveMatch
Description:	UDDI case insensitive matching find qualifier
UDDI Key (V3):	uddi:uddi.org:findqualifier:caseinsensitivematch
Derived V1,V2 format key:	uuid:9e217abc-51f0-3703-ba50-ccd9177040f0
Categorization:	findQualifier
Support:	Mandatory

11.4.3.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel
 tModelKey="uddi:uddi.org:findqualifier:caseinsensitivematch">
  <name>uddi-org:caseInsensitiveMatch</name>
  <description>UDDI case insensitive matching find qualifier
  </description>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#caseinsens
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:findQualifier"
      keyValue="findQualifier"
      tModelKey="uddi:uddi.org:categorization:types"/>
  </categoryBag>
</tModel>
```

11.4.3.4 Example of Use

The following represents a typical inquiry that references the Case Insensitive Match find qualifier tModel. All tModels that start with the name 'rosetta' are returned, without regard to case.

```
<find_tModel xmlns="urn:uddi-org:api_v3">
  <findQualifiers>
    <findQualifier>
       uddi:uddi.org:findqualifier:caseinsensitivematch
    </findQualifier>
    <findQualifier>
       uddi:uddi.org:findqualifier:approximatematch
    </findQualifier>
  </findQualifiers>
  <name>rosetta%</name>
</find_tModel>
```

11.4.4 UDDI Case Sensitive Match Find Qualifier

11.4.4.1 Introduction

The Case Sensitive Match find qualifier allows matching to occur on any inquiry involving a name, keyValue, and where relevant, keyName, where the case of the UDDI data must match case of the search argument(s).

11.4.4.2 Design Goals

The Case Sensitive Match find qualifier tModel is the implied default for both approximate and exact matching. It is provided to allow explicit specification of the case sensitivity criterion for the inquiry function.

11.4.4.3 tModel Definition

This tModel is a find qualifier that is used to specify case sensitive matching (the default) in UDDI inquiries. When this tModel is referenced in a find qualifier, case is relevant in the search results and all entries that match the value passed in the search arguments will be returned. This qualifier is ignored for languages that are not case-significant and cannot be used with the uddi-org:caseInsensitiveMatch find qualifier.

Name:	uddi-org:caseSensitiveMatch
Short name:	caseSensitiveMatch
Description:	UDDI Case Sensitive Matching find qualifier
UDDI Key (V3):	uddi:uddi.org:findqualifier:casesensitivematch
Derived V1,V2 format key:	uuid:272a64a3-73c8-3b3d-9560-c7dfaa79cc6d
Categorization:	findQualifier
Support:	Mandatory, default

11.4.4.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel
 tModelKey="uddi:uddi.org:findqualifier:casesensitivematch">
  <name>uddi-org:caseSensitiveMatch</name>
  <description>UDDI Case Sensitive Matching find qualifier
  </description>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#casesens
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:findQualifier"
      keyValue="findQualifier"
      tModelKey="uddi:uddi.org:categorization:types"/>
  </categoryBag>
</tModel>
```

11.4.4.4 Example of Use

The following represents a typical inquiry that references the Case Sensitive Match find qualifier tModel. Note that because case sensitive matching is the default for inquiry functions, specification of this find qualifier is not necessary. This example finds tModels that are categorized with a rating of AAA from the example: Rating informal category system.

```
<find_tModel xmlns="urn:uddi-org:api_v3">
  <findQualifiers>
    <findQualifier>
       uddi:uddi.org:findqualifier:casesensitivematch
    </findQualifier>
    <findQualifier>
       uddi:uddi.org:findqualifier:approximatematch
    </findQualifier>
  </findQualifiers>
  <categoryBag>
    <keyedReference
      tModelKey="uddi:uddi.org:categorization:general_keywords"
      keyName="example:Rating"
      keyValue="AAA" />
  </categoryBag>
</find_tModel>
```

11.4.5 UDDI Diacritics Insensitive Match Find Qualifier

11.4.5.1 Introduction

The Diacritics Insensitive Match find qualifier allows matching without regard to diacritics in the searched data.

11.4.5.2 Design Goals

The Diacritics Insensitive Match find qualifier tModel is provided to enable discovery of UDDI entities when diacritics sensitivity is not important.

11.4.5.3 tModel Definition

This tModel is a find qualifier that is used to enable diacritics insensitive matching in UDDI inquiries. When this tModel is referenced in a find qualifier, diacritic symbols (e.g. accents, diaeresis, cedilla) are irrelevant in the searched data and all entries, independent of diacritics, that match the base characters passed in the input criteria will be returned. This qualifier uses as diacritics, that set of characters defined to constitute diacritics in the Default Unicode Collation Element Table[46]. This qualifier is ignored for languages that are not diacritics-significant and cannot be used with the uddi-org:exactMatch or uddi-org:diacriticsSensitiveMatch find qualifiers.

Name:	uddi-org:diacriticsInsensitiveMatch
Short name:	diacriticsInsensitiveMatch
Description:	UDDI Diacritics Insensitive Matching find qualifier
UDDI Key (V3):	uddi:uddi.org:findqualifier:diacriticsinsensitivematch
Derived V1,V2 format key:	uuid:81479f4e-018a-3e8a-9ef4-2c87232b6b19
Categorization:	findQualifier
Support:	Optional

:

11.4.5.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel
 tModelKey="uddi:uddi.org:findqualifier:diacriticsinsensitivematch">
  <name>uddi-org:diacriticsInsensitiveMatch</name>
  <description>UDDI Diacritics Insensitive Matching find qualifier
  </description>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#diacritInsens
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:findQualifier"
```

[46] http://www.unicode.org/unicode/reports/tr10/allkeys.txt

```
        keyValue="findQualifier"
        tModelKey="uddi:uddi.org:categorization:types"/>
    </categoryBag>
</tModel>
```

11.4.5.4 Example of Use

The following represents a typical inquiry that references the Diacritics Insensitive Match find qualifier tModel. All tModels that start with the name '**Forte**' are returned, without regard to diacritics, even when the supplied search pattern, '**Forté**' has a diacritic.

```
<find_tModel xmlns="urn:uddi-org:api_v3">
  <findQualifiers>
    <findQualifier>
       uddi:uddi.org:findqualifier:diacriticsinsensitivematch
    </findQualifier>
    <findQualifier>
       uddi:uddi.org:findqualifier:approximatematch
    </findQualifier>
  </findQualifiers>
  <name>Forté%</name>
</find_tModel>
```

11.4.6 UDDI Diacritics Sensitive Match Find Qualifier

11.4.6.1 Introduction

The Diacritics Sensitive Match find qualifier allows matching to occur on any inquiry involving a name, keyValue, and where relevant, keyName, where the diacritics in the UDDI data must match diacritics in the search argument(s).

11.4.6.2 Design Goals

The Case Sensitive Match find qualifier tModel is the implied default for both approximate and exact matching. It is provided to allow explicit specification of the case sensitivity criterion for the inquiry function.

11.4.6.3 tModel Definition

This tModel is a find qualifier that is used to specify case sensitive matching (the default) in UDDI inquiries. When this tModel is referenced in a find qualifier, case is relevant in the search results and all entries that match the value passed in the search arguments will be returned. This qualifier is ignored for languages that are not diacritics-significant and cannot be used with uddi-org:diacriticsInsensitiveMatch find qualifier.

Name:	uddi-org:diacriticsSensitiveMatch
Short name:	diacriticsSensitiveMatch
Description:	UDDI Diacritics Sensitive Matching find qualifier
UDDI Key (V3):	uddi:uddi.org:findqualifier:diacriticssensitivematch
Derived V1,V2 format key:	uuid:ff85bfaf-1421-39a5-8207-d4df87fc2b17
Categorization:	findQualifier
Support:	Mandatory, default

11.4.6.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel
 tModelKey="uddi:uddi.org:findqualifier:diacriticssensitivematch">
  <name>uddi-org:diacriticsSensitiveMatch</name>
  <description>UDDI Diacritics Sensitive Matching find qualifier
  </description>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#diacritSens
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:findQualifier"
      keyValue="findQualifier"
      tModelKey="uddi:uddi.org:categorization:types"/>
  </categoryBag>
</tModel>
```

11.4.6.4 Example of Use

The following represents a typical inquiry that references the Diacritics Sensitive Match find qualifier tModel. Note that because diacritic sensitive matching is the default for inquiry functions, specification of this find qualifier is not necessary. This example finds tModels that have names that start with *Garçon*.

```
<find_tModel xmlns="urn:uddi-org:api_v3">
  <findQualifiers>
    <findQualifier>
       uddi:uddi.org:findqualifier:diacriticssensitivematch
    </findQualifier>
    <findQualifier>
       uddi:uddi.org:findqualifier:approximatematch
    </findQualifier>
  </findQualifiers>
  <name xml:lang="fr-ca">Garçon%</name>
</find_tModel>
```

11.4.7 UDDI Binary Sort Order Qualifier

11.4.7.1 Introduction

The sortOrder type of find qualifier (a subset of find qualifier) represents a collation sequence applied to the result set. The Binary Sort sortOrder find qualifier directs that a binary sort be performed on the result set.

11.4.7.2 Design Goals

The Binary Sort sortOrder tModel is provided to direct inquiry results to be binary sorted.

11.4.7.3 tModel Definition

This tModel is a find qualifier that is used to enable binary sorting of UDDI inquiry results. When this tModel is referenced in a find qualifier, a binary sort is performed on the name field, normalized using Unicode Normalization Form C[47]. This qualifier conflicts with any other find qualifier of the sortOrder type.

Name:	uddi-org:binarySort
Short name:	binarySort
Description:	UDDI Binary Sort collation sequence find qualifier
UDDI Key (V3):	uddi:uddi.org:sortorder:binarysort
Derived V1,V2 format key:	uuid:c0d82cac-38fe-3a0d-982c-50e0e1253eb1
Categorization:	sortOrder, findQualifier
Support:	Mandatory

11.4.7.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:sortorder:binarysort">
  <name>uddi-org:binarySort</name>
  <description>UDDI binary sort sortOrder qualifier
  </description>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#sortOrd
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:sortOrder"
      keyValue="sortOrder"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:findQualifier"
      keyValue="findQualifier"
      tModelKey="uddi:uddi.org:categorization:types"/>
  </categoryBag>
</tModel>
```

[47] http://www.unicode.org/unicode/reports/tr15/

11.4.7.4 Example of Use

The following represents a typical inquiry that references the Binary Sort sortOrder find qualifier tModel. This example finds all tModels that start with 'Rosetta', without regard to case and sorts the tModels that satisfy this criterion by date, oldest to newest, and within date, by name using a binary sorting collation sequence.

```
<find_tModel xmlns="urn:uddi-org:api_v3">
  <findQualifiers>
    <findQualifier>
       uddi:uddi.org:sortorder:binarysort
    </findQualifier>
    <findQualifier>
       uddi:uddi.org:findqualifier:sortbydateasc
    </findQualifier>
    <findQualifier>
       uddi:uddi.org:findqualifier:approximatematch
    </findQualifier>
    <findQualifier>
       uddi:uddi.org:findqualifier:caseinsensitivematch
    </findQualifier>
  </findQualifiers>
  <name>Rosetta%</name>
</find_tModel>
```

11.4.8 UDDI Unicode Technical Standard #10 Sort Order Qualifier

11.4.8.1 Introduction

The sortOrder type of find qualifier (a subset of find qualifier) represents a collation sequence applied to the result set. The Unicode Technical Standard #10 (UTS-10) sortOrder find qualifier directs that a sort be performed on the Unicode Normalization Form C[48] representation of result set elements.

11.4.8.2 Design Goals

The Unicode Technical Standard #10 sortOrder tModel is provided to enable inquiry results to be sorted according to the Unicode Technical Standard #10 Collation Order[49].

[48] http://www.unicode.org/unicode/reports/tr15/

[49] http://www.unicode.org/unicode/reports/tr10/

11.4.8.3 tModel Definition

This tModel is a find qualifier that is used to enable sorting of UDDI inquiry results based on the Unicode Technical Standard 10 Collation Order on elements normalized according to Unicode Normalization Form C. When this tModel is referenced in a find qualifier, a sort is performed according to the Default Unicode Collation Element Table in conjunction with Unicode Collation Algorithm[50] on the name field, normalized using Unicode Normalization Form C. This qualifier conflicts with any other find qualifier of the sortOrder type, such as uddi.org:binarySort:

Name:	uddi-org:UTS-10
Short name:	UTS-10
Description:	UDDI Unicode Technical Standard #10 sort collation sequence find qualifier
UDDI Key (V3):	uddi:uddi.org:sortorder:uts-10
Derived V1,V2 format key:	uuid:d93662bc-d3f5-3aab-8245-70bafc3b00dd
Categorization:	sortOrder, findQualifier
Support:	Recommended

11.4.8.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:sortorder:uts-10">
  <name>uddi-org:UTS-10</name>
  <description>UDDI Unicode Technical Standard #10 sort
              collation sequence find qualifier
  </description>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#UCASort
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:sortOrder"
      keyValue="sortOrder"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:findQualifier"
      keyValue="findQualifier"
      tModelKey="uddi:uddi.org:categorization:types"/>
  </categoryBag>
</tModel>
```

[50] http://www.unicode.org/unicode/reports/tr10/

11.4.8.4 Example of Use

The following represents a typical inquiry that references the Unicode Technical Standard #10 sortOrder find qualifier tModel. This example finds businesses or their contained businessService or bindingTemplate structures that are categorized with any value using the 'tempuri-org:CustomerType' value set. The businessEntity structures so found are sorted first by name and for those that share a common name, by date, using the Unicode Technical Standard #10 collation sequence.

```
<find_business xmlns="urn:uddi-org:api_v3">
  <findQualifiers>
    <findQualifier>
       uddi:uddi.org:sortorder:uts-10
    </findQualifier>
    <findQualifier>
       uddi:uddi.org:findqualifier:sortbydatedesc
    </findQualifier>
    <findQualifier>
       uddi:uddi.org:findqualifier:sortbynameasc
    </findQualifier>
    <findQualifier>
       uddi:uddi.org:findqualifier:approximatematch
    </findQualifier>
    <findQualifier>
       uddi:uddi.org:findqualifier:combinecategorybags
    </findQualifier>
  </findQualifiers>
  <categoryBag>
    <keyedReference keyValue="%"
      keyName="tempuri-org:CustomerType"
      tModelKey="uddi:uddi.org:categorization:general_keywords"/>
  </categoryBag>
</find_business>
```

11.4.9 UDDI Case Insensitive Sort Find Qualifier

11.4.9.1 Introduction

The Case Insensitive Sort find qualifier directs that the result set be sorted without regard to case, using the collation sequence specified.

11.4.9.2 Design Goals

The purpose of the Case Insensitive Sort find qualifier is to provide a means for requesting the results to be sorted without regard to case. For result set elements that have names, this find qualifier uses the sorting sequence in effect, but without regard to case, sorting on the first name element contained in each result element.

11.4.9.3 tModel Definition

This tModel is a find qualifier that is used to enable ordering of UDDI inquiry results using the collation sequence find qualifier in effect, but without regard to case. This sort qualifier is applied prior to any truncation of result sets and is only applicable to inquiries that return a name element in the top-most detail level of the result set. This qualifier conflicts with the uddi-org:binarySort and the uddi-org:caseSensitiveSort sort qualifiers. When a sort qualifier is also provided for the Date (uddi-org:sortByDateAsc or uddi-org:sortByDateDesc), the sort is first performed on the name fields. Results that share the same name are then sorted by date.

Name:	uddi-org: caseInsensitiveSort
Short name:	caseInsensitiveSort
Description:	UDDI sort qualifier used to sort results without regard to case
UDDI Key (V3):	uddi:uddi.org:findqualifier:caseinsensitivesort
Derived V1,V2 format key:	uuid:9240a18a-915c-3352-9112-55b3c1b2aa7f
Categorization:	findQualifier
Support:	Mandatory

11.4.9.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:findqualifier:caseinsensitivesort">
  <name>uddi-org:caseInsensitiveSort</name>
  <description>UDDI sort qualifier used to sort results without
              regard to case</description>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#caseInsensSort
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:findQualifier"
      keyValue="findQualifier"
      tModelKey="uddi:uddi.org:categorization:types"/>
   </categoryBag>
</tModel>
```

11.4.9.4 Example of Use

The following represents a typical inquiry that references the Case Insensitive Sort sort qualifier tModel. This inquiry seeks businesses that have Web service bindings that implement a RosettaNet interface and sorts the businesses returned using the Unicode Technical Standard #10 collation sequence but without regard to case.

```
<find_business xmlns="urn:uddi-org:api_v3">
  <findQualifiers>
    <findQualifier>
       uddi:uddi.org:findqualifier:caseinsensitivesort
    </findQualifier>
    <findQualifier>
       uddi:uddi.org:sortorder:uts-10
    </findQualifier>
  </findQualifiers>
  <!--find businesses that have bindings that reference
            the RosettaNet tModels -->
  <find_tModel>
    <findQualifiers>
      <findQualifier>
         uddi:uddi.org:findqualifier:approximatematch
      </findQualifier>
      <findQualifier>
         uddi:uddi.org:findqualifier:caseinsensitivematch
      </findQualifier>
    </findQualifiers>
    <name>Rosetta%</name>
  </find_tModel>
</find_business>
```

11.4.10 UDDI Case Sensitive Sort Find Qualifier

11.4.10.1 Introduction

The Case Sensitive Sort find qualifier directs that the result set be sorted according to case, using the collation sequence specified.

11.4.10.2 Design Goals

The Case Sensitive Sort find qualifier tModel is the implied default for sorting. It is provided to allow explicit specification of the case sensitivity criterion for the inquiry sorting function.

11.4.10.3 tModel Definition

This tModel is a find qualifier that is used to enable ordering of UDDI inquiry results using the collation sequence find qualifier in effect, taking case into consideration. This sort qualifier is applied prior to any truncation of result sets and is only applicable to inquiries that return a name element in the top-most detail level of the result set. This qualifier conflicts with the uddi-org:caseInsensitiveSort sort qualifier. When a sort qualifier is also provided for the Date (uddi-org:sortByDateAsc or uddi-org:sortByDateDesc), the sort is first performed on the name fields. Results that share the same name are then sorted by date.

Name:	uddi-org: caseSensitiveSort
Short name:	caseSensitiveSort
Description:	UDDI sort qualifier used to sort results considering case
UDDI Key (V3):	uddi:uddi.org:findqualifier:casesensitivesort
Derived V1,V2 format key:	uuid:437df974-faba-3428-a59e-d1550d4494a9
Categorization:	sortOrder, findQualifier
Support:	Mandatory, default

11.4.10.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:findqualifier:casesensitivesort">
  <name>uddi-org:caseSensitiveSort</name>
  <description>UDDI sort qualifier used to sort results using
    case sensitivity</description>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#caseSensSort
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:sortOrder"
      keyValue="sortOrder"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:findQualifier"
      keyValue="findQualifier"
      tModelKey="uddi:uddi.org:categorization:types"/>
   </categoryBag>
</tModel>
```

11.4.10.4 Example of Use

The following represents a typical inquiry that references the Case Sensitive Sort sort qualifier tModel. This inquiry seeks businesses that have Web service bindings that implement a RosettaNet interface and sorts the businesses returned using the Unicode Technical Standard #10 collation sequence according to case. Note that because case sensitive sorting is the default for inquiry functions, specification of this find qualifier is not necessary.

```
<find_business xmlns="urn:uddi-org:api_v3">
  <findQualifiers>
    <findQualifier>
       uddi:uddi.org:findqualifier:casesensitivesort
    </findQualifier>
    <findQualifier>
       uddi:uddi.org:sortorder:uts-10
    </findQualifier>
  </findQualifiers>
  <!--find businesses that have bindings that reference
           the RosettaNet tModels -->
  <find_tModel>
    <findQualifiers>
      <findQualifier>
         uddi:uddi.org:findqualifier:approximatematch
      </findQualifier>
      <findQualifier>
         uddi:uddi.org:findqualifier:caseinsensitivematch
      </findQualifier>
    </findQualifiers>
    <name>Rosetta%</name>
  </find_tModel>
</find_business>
```

11.4.11 UDDI Sort By Name Ascending Find Qualifier

11.4.11.1 Introduction

The Sort By Name Ascending find qualifier directs that the result set be sorted by name in ascending sequence using the collation sequence specified.

11.4.11.2 Design Goals

The purpose of the Sort By Name Ascending find qualifier is to provide a means for requesting the field (name) and directionality (ascending) with which the results are sorted. For result set elements that have names this find qualifier captures the sorting sequence in effect, sorting on the first (known as the primary) name element contained in each result element.

11.4.11.3 tModel Definition

This tModel is a find qualifier that is used to enable ordering of UDDI inquiry results using the name element from a UDDI core data structure, in an ascending sequence, as described by the collation sequence find qualifier in effect. This sort qualifier is applied prior to any truncation of result sets and is only applicable to inquiries that return a name element in the top-most detail level of the result set. When no conflicting sort qualifier is specified, this is the default sort ordering behavior. This qualifier conflicts with the Sort By Name Descending sort qualifier. When a sort qualifier is also provided for the Date (uddi-org:sortByDateAsc or uddi-org:sortByDateDesc), the sort is first performed on the name fields. Results that share the same name are then sorted by date.

Name:	uddi-org:sortByNameAsc
Short name:	sortByNameAsc
Description:	UDDI sort qualifier used to sort results by name in ascending order
UDDI Key (V3):	uddi:uddi.org:findqualifier:sortbynameasc
Derived V1,V2 format key:	uuid:8f0381c5-ef3e-3d0b-8483-5d7f480560a7
Categorization:	findQualifier
Support:	Mandatory, default

11.4.11.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:findqualifier:sortbynameasc">
  <name>uddi-org:sortByNameAsc</name>
  <description>UDDI sort qualifier used to sort results by name
               in ascending order</description>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#nameAsc
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:findQualifier"
      keyValue="findQualifier"
      tModelKey="uddi:uddi.org:categorization:types"/>
  </categoryBag>
</tModel>
```

11.4.11.4 Example of Use

The following represents a typical inquiry that references the Sort By Name Ascending sort qualifier tModel. This inquiry seeks businesses that have Web service bindings that implement a fixed interface or a RosettaNet interface and sort the businesses returned by name in ascending sequence using the Unicode Technical Standard #10 collation sequence.

```
<find_business xmlns="urn:uddi-org:api_v3">
  <findQualifiers>
    <findQualifier>
       uddi:uddi.org:findqualifier:sortbynameasc
    </findQualifier>
    <findQualifier>
       uddi:uddi.org:sortorder:uts-10
    </findQualifier>
    <findQualifier>
       uddi:uddi.org:findqualifier:orallkeys
    </findQualifier>
  </findQualifiers>
  <!--find businesses that have bindings that reference
      this fixed tModel -->
  <tModelBag>
    <tModelKey>uddi:some.specific.example:tmodelkey</tModelKey>
  </tModelBag>
  <!--OR one of the RosettaNet tModels -->
  <find_tModel xmlns="urn:uddi-org:api_v3">
    <findQualifiers>
      <findQualifier>
         uddi:uddi.org:findqualifier:approximatematch
      </findQualifier>
      <findQualifier>
         uddi:uddi.org:findqualifier:caseinsensitivematch
      </findQualifier>
      <findQualifier>
         uddi:uddi.org:sortorder:uts-10
      </findQualifier>
    </findQualifiers>
    <name>Rosetta%</name>
  </find_tModel>
</find_business>
```

11.4.12 UDDI Sort By Name Descending Find Qualifier

11.4.12.1 Introduction

The Sort By Name Descending find qualifier directs that the result set be sorted by name in descending order using the collation sequence specified.

11.4.12.2 Design Goals

The purpose of the Sort By Name Descending find qualifier is to provide a means for requesting the field (name) and directionality (descending) with which the results are sorted. For result set elements that have names this find qualifier overrides the default directionality to enable sorting on the first (known as the primary) name element contained in each result element, in a descending order.

11.4.12.3 tModel Definition

This tModel is a find qualifier that is used to enable ordering of UDDI inquiry results using the name element in each result, in a descending sequence, as described by the collation sequence find qualifier in effect. This sort qualifier is applied prior to any truncation of result sets and is only applicable to inquiries that return a name element in the top-most detail level of the result set. This qualifier conflicts with the Sort By Name Ascending sort qualifier. When a sort qualifier is also provided for the Date (uddi-org:sortByDateAsc or uddi-org:sortByDateDesc), the sort is first performed on the name fields. Results that share the same name are then sorted by date.

Name:	uddi-org:sortByNameDesc
Short name:	sortByNameDesc
Description:	UDDI sort qualifier used to sort results by name in descending order
UDDI Key (V3):	uddi:uddi.org:findqualifier:sortbynamedesc
Derived V1,V2 format key:	uuid:fa909d90-0589-3c55-bdd9-94e9bafd7e97
Categorization:	findQualifier
Support:	Mandatory

11.4.12.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel
 tModelKey="uddi:uddi.org:findqualifier:sortbynamedesc">
  <name>uddi-org:sortByNameDesc</name>
  <description>UDDI sort qualifier used to sort results by
              name in descending order</description>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#nameDesc
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:findQualifier"
      keyValue="findQualifier"
      tModelKey="uddi:uddi.org:categorization:types"/>
```

```
  </categoryBag>
</tModel>
```

11.4.12.4 Example of Use

The following represents a typical inquiry that references the Sort By Name Descending sort qualifier tModel. This inquiry seeks businesses that have Web service bindings for any WSDL-based Web Service related to Transporting goods. The businesses that are discovered are then sorted by name in descending sequence using the Unicode Technical Standard #10 collation sequence.

```
<find_business xmlns="urn:uddi-org:api_v3">
  <!--find businesses that have WSDL-based bindings related
      to Transporting Goods -->
  <findQualifiers>
    <findQualifier>
       uddi:uddi.org:findqualifier:sortbynamedesc
    </findQualifier>
    <findQualifier>
       uddi:uddi.org:sortorder:uts-10
    </findQualifier>
    <findQualifier>
       uddi:uddi.org:findqualifier:orallkeys
    </findQualifier>
  </findQualifiers>
  <!-- discover tModels that are of the WSDL type and related
       to Transporting Goods -->
  <find_tModel xmlns="urn:uddi-org:api_v3">
    <findQualifiers>
      <findQualifier>
         uddi:uddi.org:findqualifier:approximatematch
      </findQualifier>
      <findQualifier>
         uddi:uddi.org:findqualifier:orlikekeys
      </findQualifier>
      <findQualifier>
         uddi:uddi.org:sortorder:uts-10
      </findQualifier>
    </findQualifiers>
    <categoryBag>
      <keyedReference keyValue="wsdlSpec"
         tModelKey="uddi:uddi.org:categorization:types"/>
      <keyedReference keyValue="78%"
         tModelKey="uddi:uddi.org:ubr:categorization:unspsc"/>
    </categoryBag>
  </find_tModel>
</find_business>
```

First tModels of the WSDL type and are related to the Transporting Goods product category are discovered and sorted by the tModel name using the UTS-10 sort order. Then the businesses that have bindingTemplate structures that reference any of the discovered tModels are located. The resulting businesses are sorted by the name of the business in descending sequence.

11.4.13 UDDI Sort By Date Ascending Find Qualifier

11.4.13.1 Introduction

The Sort By Date Ascending find qualifier directs that the result set be sorted by the last date updated in ascending chronological order (earliest changes first) using the collation sequence specified.

11.4.13.2 Design Goals

The purpose of the Sort By Date Ascending find qualifier is to provide a means for requesting the field (last date modified) and directionality (earliest to most recent) with which the results are sorted. When this is the only sort order find qualifier specified, this find qualifier specifies sorting on the date the last update took place on each result. When a find qualifier to sort by name is also included for those result sets that contain name elements in their results, this find qualifier enables results that share a common name to be sorted by date.

11.4.13.3 tModel Definition

This tModel is a find qualifier that is used to enable ordering of UDDI inquiry results using the date last updated in each result, in an ascending sequence. See Section 3.8 *operationalInfo Structure* for more information about the date last updated. This sort qualifier is applied prior to any truncation of result sets. When the top most detail element in the result set does not contain a name element and there is no sort qualifier specified, the sort ordering behavior as defined by this find qualifier is the default sort ordering behavior. This qualifier conflicts with the Sort By Date Descending sort qualifier. Sort qualifiers involving date are secondary in precedence to the sortByName* qualifiers. Used in conjunction with one of the sortByName* qualifiers, this sort qualifier causes results to be sorted within name by date, oldest to newest.

Name:	uddi-org:sortByDateAsc
Short name:	sortByDateAsc
Description:	UDDI sort qualifier used to sort results by the last date updated in ascending order
UDDI Key (V3):	uddi:uddi.org:findqualifier:sortbydateasc
Derived V1,V2 format key:	uuid:6a5b9207-8b01-35ed-ac8e-b6502c92fc4b
Categorization:	findQualifier
Support:	Mandatory

11.4.13.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel
 tModelKey="uddi:uddi.org:findqualifier:sortbydateasc">
  <name>uddi-org:sortByDateAsc</name>
  <description>UDDI sort qualifier used to sort results by date in
               ascending order</description>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#dateAsc
    </overviewURL>
```

```
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:findQualifier"
      keyValue="findQualifier"
      tModelKey="uddi:uddi.org:categorization:types"/>
  </categoryBag>
</tModel>
```

11.4.13.4 Example of Use

The following represents a typical inquiry that references the Sort By Date Ascending sort qualifier tModel. All branches of some_company are discovered. This intersection of this set of businesses with businesses that are located in California is then sorted by date, oldest to newest.

```
<find_business xmlns="urn:uddi-org:api_v3">
  <findQualifiers>
    <findQualifier>
       uddi:uddi.org:findqualifier:sortbydateasc
    </findQualifier>
  </findQualifiers>
  <!--Find branches of some_company that are located in
      California -->
  <categoryBag>
    <keyedReference keyValue="US-CA"
        tModelKey="uddi:uddi.org:ubr:categorization:iso3166:
            business_location"/>
  </categoryBag>
  <find_relatedBusinesses xmlns="urn:uddi-org:api_v3">
    <fromKey>uddi:some_company:main_business</fromKey>
    <keyedReference keyValue="parent-child"
        keyName="branch"
        tModelKey="uddi:uddi.org:ubr:relationships"/>
  </find_relatedBusinesses>
</find_business>
```

11.4.14 UDDI Sort By Date Descending Find Qualifier

11.4.14.1 Introduction

The Sort By Date Descending find qualifier directs that the result set be sorted by the last date updated in descending chronological order (most recent changes first) using the collation sequence specified.

11.4.14.2 Design Goals

The purpose of the Sort By Date Descending find qualifier is to provide a means for requesting the field and directionality with which the results are sorted. This find qualifier specifies sorting on the date the last update took place on the element in descending order (newest to oldest). When a find qualifier to sort by name is also included for those result sets that contain name elements in their results, this find qualifier enables results that share a common name to be sorted by date.

11.4.14.3 tModel Definition

This tModel is a find qualifier that is used to enable ordering of UDDI inquiry results using the date last updated for each result, in a descending sequence. See Section 3.8 *operationalInfo* Structure for more information about the date last updated. This sort qualifier is applied prior to any truncation of result sets. This qualifier conflicts with the Sort By Date Ascending sort qualifier. Sort qualifiers involving date are secondary in precedence to the sortByName qualifiers. Used in conjunction with one of the sortByName qualifiers, this sort qualifier causes results to be sorted within name by date, newest to oldest.

Name:	uddi-org:sortByDateDesc
Short name:	sortByDateDesc
Description:	UDDI sort qualifier used to sort results by the date last updated in descending order
UDDI Key (V3):	uddi:uddi.org:findqualifier:sortbydatedesc
Derived V1,V2 format key:	uuid:d325de47-150c-3446-a646-b43fbeaf6905
Categorization:	findQualifier
Support:	Mandatory

11.4.14.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel
 tModelKey="uddi:uddi.org:findqualifier:sortbydatedesc">
  <name>uddi-org:sortByDateDesc</name>
  <description>UDDI sort qualifier used to sort results
              by date in descending order</description>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#dateDesc
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:findQualifier"
```

```
        keyValue="findQualifier"
        tModelKey="uddi:uddi.org:categorization:types"/>
    </categoryBag>
</tModel>
```

11.4.14.4 Example of Use

The following represents a typical inquiry that references the Sort By Date Descending sort qualifier tModel. This inquiry locates all of the approved value set validation Web services when the registry follows the recommended policy and sorts the tModels with the most recent ones first.

```
<find_business xmlns="urn:uddi-org:api_v3">
  <findQualifiers>
    <findQualifier>
       uddi:uddi.org:findqualifier:sortbydatedesc
    </findQualifier>
    <findQualifier>
       uddi:uddi.org:findqualifier:approximatematch
    </findQualifier>
    <findQualifier>
       uddi:uddi.org:findqualifier:orallkeys
    </findQualifier>
  </findQualifiers>
  <!-- find approved validation services within the registry's node
       businessEntities -->
  <tModelBag>
     <tModelKey>uddi:uddi.org:v3_checkedvalueset</tModelKey>
  </tModelBag>
  <identifierBag>
    <keyedReference keyValue="%"
      tModelKey="uddi:uddi.org:identifier:nodes"/>
  </identifierBag>
</find_business>
```

Node Business Entity elements are discovered that contain one or more descendent bindingTemplate elements that implement the validate_values API. These businesses are then sorted by date in descending sequence.

11.4.15 UDDI And All Keys Find Qualifier

11.4.15.1 Introduction

The And All Keys find qualifier directs that a logical AND be performed on any bag contents prior to performing the specified search.

11.4.15.2 Design Goals

The design goal of the And All Keys find qualifier is to provide a means for finding entities that have all of the values specified, without regard to the value set that they are associated with. For some bags this find qualifier merely enforces default behavior (categoryBag and tModelBag). For identifierBags, this find qualifier overrides the default behavior.

11.4.15.3 tModel Definition

This tModel is referenced by a find qualifier in a UDDI inquiry API to AND the keys in identifierBag, categoryBag, and/or tModelBag prior to performing the actual search for results. Using this find qualifier tModel does not change the treatment of categoryBag or tModelBag contents because ANDing their bag contents is default behavior.

This find qualifier cannot be used with the uddi-org:orAllKeys or uddi-org:orLikeKeys find qualifier. The set of values that would result from use of a wildcard in a keyValue are always implicitly OR'd together independent of and after application of this wildcard. This allows an inquiry of the form 'find businesses with services in Georgia that relate to transporting goods', where Georgia would be specified as an explicit keyValue from a geographic value set and transporting goods would be specified using a partial value with a wildcard from a product and services value set.:

Name:	uddi-org:andAllKeys
Short name:	andAllKeys
Description:	UDDI find qualifier used to request that a logical AND be performed on bag contents prior to a search.
UDDI Key (V3):	uddi:uddi.org:findqualifier:andallkeys
Derived V1,V2 format key:	uuid:c2ec3b53-7bac-3f59-b313-61882e417cde
Categorization:	findQualifier
Support:	Mandatory

11.4.15.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel
 tModelKey="uddi:uddi.org:findqualifier:andallkeys">
  <name>uddi-org:andAllKeys</name>
  <description>UDDI find qualifier used to request that a
               logical AND be performed on bag contents
               prior to a search</description>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#andAll
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:findQualifier"
      keyValue="findQualifier"
      tModelKey="uddi:uddi.org:categorization:types"/>
   </categoryBag>
</tModel>
```

11.4.15.4 Example of Use

The following represents a typical inquiry that references the And All Keys find qualifier tModel. This inquiry locates the tModels that are owned by some_company and which have a D-U-N-S® identity.

```
<find_business xmlns="urn:uddi-org:api_v3">
  <findQualifiers>
    <findQualifier>
       uddi:uddi.org:findqualifier:andallkeys
    </findQualifier>
    <findQualifier>
       uddi:uddi.org:findqualifier:approximatematch
    </findQualifier>
  </findQualifiers>
  <identifierBag>
    <keyedReference keyValue="uddi:some_company:main_business"
      tModelKey="uddi:uddi.org:ubr:identifier:owningbusiness"/>
    <keyedReference keyValue="%"
      tModelKey="uddi:uddi.org:ubr:identifier:dnb.com:d-u-n-s"/>
  </identifierBag>
</find_business>
```

11.4.16 UDDI Or All Keys Find Qualifier

11.4.16.1 Introduction

The Or All Keys find qualifier directs that a logical OR be performed on any bag contents prior to performing the specified search.

11.4.16.2 Design Goals

The design goal of the Or All Keys find qualifier is to provide a means for finding entities that reference any one of a set of values, without regard to the value set that they are associated with. For some bags this find qualifier merely enforces default behavior (identifierBag). For categoryBags and tModelBags, this find qualifier overrides the default behavior.

11.4.16.3 tModel Definition

This tModel is referenced by a find qualifier in a UDDI inquiry API to OR the keys in identifierBag, categoryBag, and/or tModelBag prior to performing the actual search for results. Using this find qualifier tModel does not change the treatment of identifierBag contents because ORing the contents of this bag is default behavior. This find qualifier cannot be used with the uddi-org:andAllKeys or uddi-org:orLikeKeys find qualifiers.

Name:	uddi-org:orAllKeys
Short name:	orAllKeys
Description:	UDDI find qualifier used to request that a logical OR be performed on bag contents prior to a search.
UDDI Key (V3):	uddi:uddi.org:findqualifier:orallkeys
Derived V1,V2 format key:	uuid:f5f07764-50ac-378e-9fdd-a42a87329838
Categorization:	findQualifier

Support: Mandatory

11.4.16.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:findqualifier:orallkeys">
  <name>uddi-org:orAllKeys</name>
  <description>UDDI find qualifier used to request that a
               logical OR be performed on bag contents
               prior to a search</description>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#orAll
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:findQualifier"
      keyValue="findQualifier"
      tModelKey="uddi:uddi.org:categorization:types"/>
   </categoryBag>
</tModel>
```

11.4.16.4 Example of Use

The following represents a typical inquiry that references the Or All Keys find qualifier tModel. All businesses that are themselves categorized or contain descendents that are categorized in some way as 'Witch-doctors or voodoo services' are returned. The Or All Keys find qualifier is used to find references to this product category as native references, or as a part of some group of references.

```
<find_business xmlns="urn:uddi-org:api_v3">
  <!--Find Businesses or their services that are categorized
      in some way as Witch-doctors or voodoo services -->
  <findQualifiers>
    <findQualifier>
       uddi:uddi.org:findqualifier:orallkeys
    </findQualifier>
    <findQualifier>
       uddi:uddi.org:findqualifier:combinecategorybags
    </findQualifier>
  </findQualifiers>
  <categoryBag>
    <keyedReference keyValue="85.14.15.01.00"
        tModelKey="uddi:uddi.org:ubr:categorization:unspsc"/>
    <keyedReferenceGroup
      tModelKey="uddi:uddi.org:ubr:categorizationgroup:
                 iso3166_unspsc">
      <keyedReference keyValue="85.14.15.01.00"
        tModelKey="uddi:uddi.org:ubr:categorization:unspsc"/>
    </keyedReferenceGroup>
  </categoryBag>
</find_business>
```

11.4.17 UDDI Or Like Keys Find Qualifier

11.4.17.1 Introduction

The Or Like Keys find qualifier directs that a logical OR be performed on any bag contents that share the same value set, as specified by the tModelKeys, and an AND be perform between designated value sets prior to performing the specified search.

11.4.17.2 Design Goals

The design goal of the Or Like Keys find qualifier is to provide a means for finding entities that have at least one of the designated values from each of value sets referenced. This find qualifier allows entities to be discovered that have been described with different levels of specificity in their categoryBags and identifierBags.

11.4.17.3 tModel Definition

This tModel is referenced by a find qualifier in a UDDI inquiry API to OR the keys in categoryBag, and/or identifierBags that are from the same value set, and AND the referenced value sets prior to performing the actual search for results (i.e., one of the values A, B, or C from value set V1 AND one of the values X, Y, or Z form value set V2). The behavior of keyedReferenceGroups is analogous to that of keyedReferences, that is, keyedReferenceGroups that have the same tModelKey value are OR'd together rather than AND'd. This find qualifier conflicts with the uddi-org:andAllKeys and uddi-org:orAllKeys find qualifiers:

Name:	uddi-org:orLikeKeys
Short name:	orLikeKeys
Description:	UDDI find qualifier used to find entities that reference one of the values from each referenced value set.
UDDI Key (V3):	uddi:uddi.org:findqualifier:orlikekeys
Derived V1,V2 format key:	uuid:ba01e855-5234-3d88-a84e-b878e37b505c
Categorization:	findQualifier
Support:	Mandatory

11.4.17.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:findqualifier:orlikekeys">
  <name>uddi-org:orLikeKeys</name>
  <description>UDDI find qualifier used to find entities
              that reference one of the values from each
              referenced value set</description>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#orLike
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:findQualifier"
      keyValue="findQualifier"
```

```
      tModelKey="uddi:uddi.org:categorization:types"/>
   </categoryBag>
</tModel>
```

11.4.17.4 Example of Use

The following represents a typical inquiry that references the Or Like Keys find qualifier tModel. This example seeks bridal clothing (Tuxedo or formal wear (mens) or Evening or bridal gown (womens)) rental stores (clothing stores) that serve California.

```
<find_business xmlns="urn:uddi-org:api_v3">
  <!--Find clothing stores that serve California which
      offer rental of wedding attire.  This is represented
      in an industry, a service location, and two product
      and services categorizations.  The inquiry is
      characterized as serves California AND is any clothing
      store AND rents Tuxedos OR bridal gowns.-->
  <findQualifiers>
    <findQualifier>
       uddi:uddi.org:findqualifier:orlikekeys
    </findQualifier>
    <findQualifier>
       uddi:uddi.org:findqualifier:combinecategorybags
    </findQualifier>
    <findQualifier>
       uddi:uddi.org:findqualifier:approximatematch
    </findQualifier>
  </findQualifiers>
  <categoryBag>
    <!--Any kind of clothing store -->
    <keyedReference keyValue="7810%"
      tModelKey="uddi:uddi.org:ubr:categorization:naics:1997"/>
    <!--Tuxedo or formal wear rental -->
    <keyedReference keyValue="91.10.18.01.00"
      tModelKey="uddi:uddi.org:ubr:categorization:unspsc"/>
    <!--Evening or bridal gown or dress rental -->
    <keyedReference keyValue="91.10.18.02.00"
      tModelKey="uddi:uddi.org:ubr:categorization:unspsc"/>
    <!--Serving California, USA -->
    <keyedReference keyValue="US-CA"
      tModelKey="uddi:uddi.org:ubr:categorization:iso3166"/>
  </categoryBag>
</find_business>
```

This example finds clothing stores that serve California which offer rental of wedding attire. This is represented using an industry categorization, a service location categorization, and two product and services categorizations. The inquiry is characterized using the categoryBag contents and the OrLikeKeys findQualifier as

- is any clothing store AND
- rents tuxedos OR bridal gowns AND
- serves California.

Because the ‘Tuxedo or formal wear rental’ and the ‘Evening or bridal gown or dress rental’ keyedReference elements are from the same value set as described by their tModelKeys, the OrLikeKeys find qualifier allows businesses to be found that reference either of these product categorizations.

11.4.18 UDDI Combine Category Bags Find Qualifier

11.4.18.1 Introduction

The Combine Category Bags find qualifier, applied to the find_business inquiry, combines the categoryBag contents of each businessEntity with those of its businessService structures to enable discovery based on categorization that is independent of where the publishers placed such categorizations.

11.4.18.2 Design Goals

The design goal of the Combine Category Bags find qualifier is to provide a means for finding entities without regard to placement of the categoryBags in the published entities. This find qualifier enables inquiries involving categorization to be successful independent of the modeling styles of the publishers. Some publishers may place certain categorizations at the business level, such as those related to the industries that their businesses operate in, and place others at the service level (product and service, for example). Inquirers may not care whether the categorization is at the business level or the service level. This find qualifier enables such inquiries.

11.4.18.3 tModel Definition

This tModel is referenced by a find qualifier in a UDDI find_business inquiry API to treat the categoryBag contents of businessEntity structures and contained businessService structures as one. Searching for a category will yield a positive match on a registered business if any of the categoryBag elements contained within the businessEntity element or any of its contained or referenced businessService elements contains the filter criteria.

This find qualifier overrides the default behavior of find_business which performs matching on categoryBags on the businessEntity structures themselves by instead performing matching on categoryBags associated with contained businessService structures as well as the businessEntity structures. The uddi-org:serviceSubset and uddi-org:bindingSubset find qualifiers conflict with this one.

Name:	uddi-org:combineCategoryBags
Short name:	combineCategoryBags
Description:	UDDI find qualifier used to treat all of the categoryBags within a businessEntity as if they were one during an inquiry.
UDDI Key (V3):	uddi:uddi.org:findqualifier:combinecategorybags
Derived V1,V2 format key:	uuid:58da61de-bd1e-3ea9-8796-2ec4bfcc15fc
Categorization:	findQualifier
Support:	Mandatory

11.4.18.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel
 tModelKey="uddi:uddi.org:findqualifier:combinecategorybags">
```

```
  <name>uddi-org:combineCategoryBags</name>
  <description>UDDI find qualifier used to treat all of the
               categoryBags within a businessEntity as if
               they were one during inquiry</description>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#combineCatBags
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:findQualifier"
      keyValue="findQualifier"
      tModelKey="uddi:uddi.org:categorization:types"/>
   </categoryBag>
</tModel>
```

11.4.18.4 Example of Use

The following represents a typical inquiry that references the Combine Category Bags find qualifier tModel. This example is used to locate businesses that are categorized as 'computer facilities management services' or that have descendents that are so categorized.

```
<find_business xmlns="urn:uddi-org:api_v3">
  <!--Find computer facilities management services -->
  <findQualifiers>
    <findQualifier>
        uddi:uddi.org:findqualifier:combinecategorybags
    </findQualifier>
  </findQualifiers>
  <categoryBag>
    <keyedReference keyValue="541513"
      tModelKey="uddi:uddi.org:ubr:categorization:naics:1997"/>
  </categoryBag>
</find_business>
```

11.4.19 UDDI Service Subset Find Qualifier

11.4.19.1 Introduction

The Service Subset find qualifier, applied to the find_business inquiry, causes only categoryBags on contained businessService structures to be inspected. The categoryBags on businessEntity structures themselves are ignored.

11.4.19.2 Design Goals

Publishers often categorize their business services with the industries, products, service types and geographical service areas that apply specifically to the described services, and categorize the containing businessEntity structures with supersets of categorizations from the contained businessService structures. The design goal of the Service Subset find qualifier is to provide a means for finding business services using categorization using the find_business inquiry API, omitting those businessEntity structures that may offer the kinds of services, but which have not specifically identified them as meeting the specified needs.

11.4.19.3 tModel Definition

This tModel is referenced by a find qualifier in a UDDI find_business inquiry API to ignore the categoryBag contents of businessEntity structures and inspect only the categoryBags of contained businessService structures. Searching with a category yields a match on a registered business if any of the categoryBag elements associated with contained businessService structures contain the search category.

This find qualifier overrides the default behavior of find_business which performs matching on categoryBags on the businessEntity structures themselves. The uddi-org:combineCategoryBags and uddi-org:bindingSubset find qualifiers conflict with this one.

Name:	uddi-org:serviceSubset
Short name:	serviceSubset
Description:	UDDI find qualifier used to use categoryBags of businessService elements to satisfy the find_business inquiry.
UDDI Key (V3):	uddi:uddi.org:findqualifier:servicesubset
Derived V1,V2 format key:	uuid:55913724-e5ce-3188-9b73-4486f1ac7cd9
Categorization:	findQualifier
Support:	Mandatory

:

11.4.19.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel
 tModelKey="uddi:uddi.org:findqualifier:servicesubset">
  <name>uddi-org:serviceSubset</name>
  <description>UDDI find qualifier used to use categoryBags
               of businessService elements to satisfy the
               find_business inquiry.</description>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#servSubset
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:findQualifier"
      keyValue="findQualifier"
      tModelKey="uddi:uddi.org:categorization:types"/>
   </categoryBag>
</tModel>
```

11.4.19.4 Example of Use

The following represents a typical inquiry that references the Service Subset find qualifier tModel. This inquiry returns businessEntity elements that have descendent businessService elements that are categorized with 'Customer Computer Programming Services' or 'Computer Systems Design Services' and which conform to some technical fingerprint.

```
<find_business xmlns="urn:uddi-org:api_v3">
  <findQualifiers>
    <findQualifier>
       uddi:uddi.org:findqualifier:servicesubset
    </findQualifier>
    <findQualifier>
       uddi:uddi.org:findqualifier:orallkeys
    </findQualifier>
  </findQualifiers>
  <tModelBag>
    <tModelKey>uddi:some.specific.example:tmodelkey</tModelKey>
  </tModelBag>
  <categoryBag>
    <keyedReference keyValue="541511"
      tModelKey="uddi:uddi.org:ubr:categorization:naics:1997"/>
    <keyedReference keyValue="541512"
      tModelKey="uddi:uddi.org:ubr:categorization:naics:1997"/>
  </categoryBag>
</find_business>
```

11.4.20 UDDI Binding Subset Find Qualifier

11.4.20.1 Introduction

The Binding Subset find qualifier, applied to the find_business and find_service inquiries, cause only categoryBags on descendent bindingTemplate structures to be inspected. The categoryBags on businessEntity and/or businessService elements themselves are ignored.

11.4.20.2 Design Goals

Publishers can categorize their bindingTemplate elements with categories to distinguish between similar bindingTemplate structures. Such bindingTemplate structures may represent the same Web service but for different audiences or may represent different levels of release, for example. The design goal of the Binding Subset find qualifier is to provide a means for finding businessEntity elements or businessService elements that have bindingTemplate structures categorized in a particular way.

11.4.20.3 tModel Definition

This tModel is referenced as a find qualifier in a UDDI find_business or find_service inquiry API to ignore the categoryBag contents of containing businessEntity and businessService elements and inspect only the categoryBags of contained bindingTemplate elements. Searching with a category yields a match on a registered business if any of the categoryBag elements associated with contained bindingTemplate structures belong to the search category.

This find qualifier overrides the default behavior of find_business which performs matching on categoryBags on the businessEntity elements themselves, and of find_service which performs matching on categoryBags on the businessService elements themselves. The uddi-org:combineCategoryBags and uddi-org:serviceSubset find qualifiers conflict with this one.

Name:	uddi-org:bindingSubset
Short name:	bindingSubset
Description:	UDDI find qualifier for specifying use of categoryBags of bindingTemplate elements to satisfy the find_business or find_service inquiries.
UDDI Key (V3):	uddi:uddi.org:findqualifier:bindingsubset
Derived V1,V2 format key:	uuid:81b2ec2b-e72e-31f6-8fce-7368a0d5a671
Categorization:	findQualifier
Support:	Mandatory

11.4.20.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:findqualifier:bindingsubset">
  <name>uddi-org:bindingSubset</name>
  <description>UDDI find qualifier for specifying use of
               categoryBags of bindingTempate elements to satisfy
               the find_business or find_service inquiries.
  </description>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#bindSubset
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:findQualifier"
      keyValue="findQualifier"
      tModelKey="uddi:uddi.org:categorization:types"/>
   </cagegoryBag>
</tModel>
```

11.4.20.4 Example of Use

The following represents a typical inquiry that references the Binding Subset find qualifier tModel. This inquiry returns businessEntity elements that have descendent bindingTemplate elements that are categorized with the 'Retail' Customer Type and which conform to some technical fingerprint.

```
<find_business xmlns="urn:uddi-org:api_v3">
  <findQualifiers>
    <findQualifier>
       uddi:uddi.org:findqualifier:bindingsubset
    </findQualifier>
    <findQualifier>
       uddi:uddi.org:findqualifier:caseinsensitivematch
    </findQualifier>
  </findQualifiers>
  <tModelBag>
    <tModelKey>uddi:some.specific.example:tmodelkey</tModelKey>
  </tModelBag>
  <categoryBag>
    <keyedReference keyValue="Retail"
      keyName="tempuri-org:CustomerType"
      tModelKey="uddi:uddi.org:categorization:general_keywords"/>
  </categoryBag>
</find_business>
```

11.4.21 UDDI Suppress Projected Services Find Qualifier

11.4.21.1 Introduction

The Suppress Projected Services find qualifier, applied to the find_business or find_service inquiry, is used to exclude service projections from all matching comparisons and from the list of serviceInfos included in businessInfos.

11.4.21.2 Design Goals

The design goal of the Suppress Projected Services find qualifier is to provide a means for excluding projected services from appearing in result sets when such appearance is unnecessary or ambiguous. When find_service is issued with no businessKey, this find qualifier is implicitly in effect because without knowledge of the containing businessEntity, service projections appear as exact duplicates of the services on which they project. This find qualifier can be used to only include actual implementations of a service, excluding those that just refer to these concrete implementations.

11.4.21.3 tModel Definition

This tModel is referenced by a find qualifier in a UDDI find_business or find_service inquiry API to ignore service projections when performing the matching algorithm and when building the serviceInfos to include in the result set. The behavior associated with this find qualifier is to ignore service projections in all stages of the inquiry, as if they are not there at all.

When a UDDI inquiry couples this find qualifier with the use of tModelBags or the serviceSubset find qualifier, empty outer elements (businessInfo for find_business; serviceInfo for find_service) are excluded from the result set when the only qualifying entity corresponds to a suppressed projected service.

This find qualifier overrides the default behavior of find_business and find_service (when a businessKey is provided).

Name:	uddi-org:suppressProjectedServices
Short name:	suppressProjectedServices
Description:	UDDI find qualifier used to exclude service projections from an inquiry function at all levels.
UDDI Key (V3):	uddi:uddi.org:findqualifier:suppressprojectedservices
Derived V1,V2 format key:	uuid:805e73a2-dcc0-3c1b-bfd8-66c5af40fc98
Categorization:	findQualifier
Support:	Mandatory

:

11.4.21.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel
   tModelKey="uddi:uddi.org:findqualifier:
             suppressProjectedServices">
  <name>uddi-org:suppressProjectedServices</name>
  <description>UDDI find qualifier used to exclude service
              projections from an inquiry function at all
              levels.</description>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#suppressProjSvcs
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:findQualifier"
      keyValue="findQualifier"
      tModelKey="uddi:uddi.org:categorization:types"/>
   </categoryBag>
</tModel>
```

11.4.21.4 Example of Use

The following represents a typical inquiry that references the Suppress Projected Services find qualifier tModel. The businesses that are the actual providers of a business service that adheres to some technical fingerprint are returned. Other businesses that refer to such businessService elements using service projections are excluded.

```
<find_business xmlns="urn:uddi-org:api_v3">
  <!--Find actual implementations of some technical
     interface -->
  <findQualifiers>
    <findQualifier>
       uddi:uddi.org:findqualifier:suppressprojectedservices
    </findQualifier>
  </findQualifiers>
  <tModelBag>
    <tModelKey>
      uddi:some.specific.example:tmodelkey
    </tModelKey>
  </tModelBag>
```

```
</find_business>
```

11.4.22 UDDI Signature Present Find Qualifier

11.4.22.1 Introduction

The Signature Present find qualifier filters the result set to include only entities that are signed or whose containing entities are signed.

11.4.22.2 Design Goals

The design goal of the Signature Present find qualifier is to provide a starting point for establishing some confidence in the entities returned from an inquiry. Only elements that are covered by XML Digital Signatures are returned. The actual signature(s) is obtained by subsequent get_xx API calls and can be verified by the client.

11.4.22.3 tModel Definition

This tModel is referenced by a find qualifier in a UDDI inquiry API to only include UDDI entities that have XML Digital Signatures or that are descendents of UDDI entities that have XML Digital Signatures. To be returned, either the elements themselves must be signed, or at least one of the elements that contain them must be signed. For example when the Signature Present find qualifier is used with the find_business inquiry API, only businessEntity elements that have an XML Digital Signature are returned. Similarly, when Signature Present is used with the find_service inquiry API, only businessService elements that have an XML Digital Signature or whose containing businessEntity elements have an XML Digital Signature are returned.

The UDDI Inquiry API provides no verification associated with use of this find qualifier.

Name:	uddi-org:signaturePresent
Short name:	signaturePresent
Description:	UDDI find qualifier used to return only entities that have or are contained in entities that have XML Digital Signatures.
UDDI Key (V3):	uddi:uddi.org:findqualifier:signaturepresent
Derived V1,V2 format key:	uuid:8a16ea11-6f8f-3085-a467-576f89f129c8
Categorization:	findQualifier
Support:	Mandatory

11.4.22.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:findqualifier:signaturepresent">
  <name>uddi-org:signaturePresent</name>
  <description>UDDI findQualifier used to return only entities
               that have or are contained in entities that have
               XML Digital Signatures.</description>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#sign
    </overviewURL>
```

```
   </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:findQualifier"
      keyValue="findQualifier"
      tModelKey="uddi:uddi.org:categorization:types"/>
   </categoryBag>
</tModel>
```

11.4.22.4 Example of Use

The following represents a typical inquiry that references the Is Signed find qualifier tModel.

```
<find_business xmlns="urn:uddi-org:api_v3">
  <tModelBag>
    <tModelKey>
         uddi:some.specific.example:tmodelkey
         </tModelKey>
  </tModelBag>
  <find_relatedBusinesses xmlns="urn:uddi-org:api_v3">
    <findQualifiers>
      <findQualifier>
          uddi:uddi.org:findqualifier:signaturepresent
      </findQualifier>
    </findQualifiers>
    <fromKey>
          uddi:some_trust_certifier.example:main_business
    </fromKey>
  </find_relatedBusinesses>
</find_business>
```

This example performs a two phase inquiry. First, businesses that have a signed a relationship with some_trust_certifier are located using the embedded find_relatedBusinesses. An intersection is built between this set of businesses and the businesses that offer Web services compliant with the technical fingerprint captured in the tModelBag. Note that there is no verification that the signatures associated with the returned relationships is valid prior to obtaining businesses that have Web services that satisfy the technical fingerprint part of the inquiry.

11.5 Other Canonical tModels

The following tModels round out the set of tModels UDDI registries MUST contain.

There are other tModels, beyond those listed here, which are defined to help classify elements within leading industry encoding schemes and standard protocols. These other tModels are maintained by the UDDI Business Registry and the list is expected to expand as appropriate as this registry expands. See the UDDI Business Registry for these tModels.

11.5.1 Domain Key Generator for the UDDI Domain

11.5.1.1 Introduction

For UDDI tModels to have publisher assigned keys, a key generator tModel for the UDDI domain must be registered and owned by UDDI itself. Ownership of such a domain key generator tModel allows UDDI owned entities, such as tModels, to have keys derived from the UDDI domain. All tModels contain in this chapter have such keys. This tModel establishes the UDDI key generator domain. Other publishers that wish to provide their own keys for UDDI core entities must similarly register their own domain key generator tModels.

11.5.1.2 Design Goals

The design goal for the UDDI Domain Key Generator tModel is to establish a key generator domain within a UDDI registry that allows UDDI to publish its canonical tModels. All other tModels described in this document have keys derived from the key generator domain established by this tModel.

11.5.1.3 tModel Definition

Name:	uddi-org:keyGenerator
Description:	UDDI domain key generator
UDDI Key (V3):	uddi:uddi.org:keygenerator
Derived V1,V2 format key:	uuid:dfd2e89d-59c1-3921-822c-da6a8f6ef57e
Categorization:	keyGenerator

11.5.1.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:keygenerator" xmlns="urn:uddi-org:api_v3">
       <name>uddi-org:keyGenerator</name>
       <description>UDDI domain key generator</description>
       <overviewDoc>
            <overviewURL
useType="text">http://uddi.org/pubs/uddi_v3.htm#keyGen</overviewURL>
       </overviewDoc>
       <categoryBag>
            <keyedReference tModelKey="uddi:uddi.org:categorization:types"
keyName="uddi-org:types:keyGenerator" keyValue="keyGenerator"/>
       </categoryBag>
       <n0:Signature xmlns:n0="http://www.w3.org/2000/09/xmldsig#">
            <n0:SignedInfo>
                <n0:CanonicalizationMethod Algorithm="urn:uddi-
org:schemaCentricC14N:2002-07-10"/>
```

```
                <n0:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-
sha1"/>
                <n0:Reference URI="">
                    <n0:Transforms>
                        <n0:Transform
Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
                        <n0:Transform Algorithm="urn:uddi-org:schemaCentricC14N:2002-
07-10"/>
                    </n0:Transforms>
                    <n0:DigestMethod
Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
                    <n0:DigestValue>LeutnHMMoAlgmZB2O+CA0gGRYOg=</n0:DigestValue>
                </n0:Reference>
            </n0:SignedInfo>

       <n0:SignatureValue>HqO3W3y3Sol8PxR6Eus1VGxrCB+mVj9A9+zWs4YbQZWU/mIGhfR2WelT67QP
spCm4LqUu1sijKI3crhYlsV1gmdFjeSQQi9ZdgIjkkKPNprW8GMqDAGc0Fl+iYFlnm80Dw5Pf7OhjR9P4S9rJaU
C4PiKjc3hNBUH4hFER++Elzw=</n0:SignatureValue>
            <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
                <KeyValue>
                    <RSAKeyValue>

       <Modulus>k0b36DMw/GfWHOA7u0kFSlH5CkYkyKs84jss2frIRRiJnH84cjVwAatm/HwGEhP5A0Dxis
4I1ZMmA85Pni+awoWHeqZEjgwmCQCRg+hM+jR1sduzsP1LOjDXre7obetIh/OHXzi/4Rb9Sx4sWBtMhwlsLMbB0
aj05PiGQuwgvgE=</Modulus>
                        <Exponent>AQAB</Exponent>
                    </RSAKeyValue>
                </KeyValue>
                <X509Data>
                    <X509IssuerSerial>
                        <X509IssuerName>OU=Secure Server Certification Authority,O=RSA
Data Security\, Inc.,C=US</X509IssuerName>

       <X509SerialNumber>167897432056789938398776780754668806215</X509SerialNumber>
                    </X509IssuerSerial>
                    <X509SubjectName>CN=uddi.org,OU=OASIS UDDI Specification TC,O=OASIS
Open,L=Billerica,ST=Massachusetts,C=US</X509SubjectName>
<X509Certificate>MIID0jCCAz+gAwIBAgIQfk/eg8kf2LIWOlLE/xzgRzANBgkqhkiG9w0BAQUFADBfMQswCQ
YDVQQGEwJVUzEgMB4GA1UEChMXUlNBIERhdGEgU2VjdXJpdHksIEluYy4xLjAsBgNVBAsTJVNlY3VyZSBTZXJ2Z
XIgQ2VydGlmaWNhdGlvbiBBdXRob3JpdHkwHhcNMDQxMTA0MDAwMDAwWhcNMDUxMTA0MjM1OTU5WjCBhzELMAkG
A1UEBhMCVVMxFjAUBgNVBAgTDU1hc3NhY2h1c2V0dHMxEjAQBgNVBAcUCUJpbGxlcmljYTETMBEGA1UEChQKT0F
TSVMgT3BlbjEkMCIGA1UECxQbT0FTSVMgVURESSBTcGVjaWZpY2F0aW9uIFRDMREwDwYDVQQDFAh1ZGRpLm9yZz
CBnzANBgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEAk0b36DMw/GfWHOA7u0kFSlH5CkYkyKs84jss2frIRRiJnH84c
jVwAatm/HwGEhP5A0Dxis4I1ZMmA85Pni+awoWHeqZEjgwmCQCRg+hM+jR1sduzsP1LOjDXre7obetIh/OHXzi/
4Rb9Sx4sWBtMhwlsLMbB0aj05PiGQuwgvgECAwEAAaOCAWgwggFkMAkGA1UdEwQCMAAwCwYDVR0PBAQDAgWgMEA
GA1UdHwQ5MDcwNaAzoDGGL2h0dHA6Ly9TVlJTZWN1cmUtY3JsLnZlcmlzaWduLmNvbS9TVlJTZWN1cmUuY3JsME
QGA1UdIAQ9MDswOQYLYIZIAYb4RQEHFwMwKjAoBggrBgEFBQcCARYcaHR0cHM6Ly93d3cudmVyaXNpZ24uY29tL
3JwYTAdBgNVHSUEFjAUBggrBgEFBQcDAQYIKwYBBQUHAwIwbQYIKwYBBQUHAQwEYTBfoV2gWzBZMFcwVRYJaW1h
Z2UvZ2lmMCEwHzAHBgUrDgMCGgQUj+XTGoasjY5rw8+AatRIGCx7GS4wJRYjaHR0cDovL2xvZ28udmVyaXNpZ24
uY29tL3ZzbG9nby5naWYwNAYIKwYBBQUHAQEEKDAmMCQGCCsGAQUFBzABhhhodHRwOi8vb2NzcC52ZXJpc2lnbi
5jb20wDQYJKoZIhvcNAQEFBQADfgAfwogDewF6ySqxv/QYtWqzoQcL8wr9oFdJawYLW0WEHxwJztv2dfbMl7/wY
wlxua6qlXc39CCdZTvkKwike4aAyLlg08ZQlvbIoPIIMPfKRJWbL6PIJSjtCdn8FFUY+LdQBWSAsVeTkMI7YK3s
6PPxpIPXDNrpaKX3NHVFWg==</X509Certificate>
                </X509Data>
            </KeyInfo>
        </n0:Signature>
</tModel>
```

11.5.2 Key Generator for UDDI Categorization tModels

11.5.2.1 Introduction

This tModel establishes the UDDI categorization key partition. This key partition is used for the categorization tModels in this specification.

11.5.2.2 Design Goals

The design goal for the Key Generator for UDDI Categorization tModels is to establish a key partition within a UDDI registry that allows UDDI to publish its canonical categorization tModels. All categorization tModels described in this document have keys derived in the key partition associated with this key generator.

11.5.2.3 tModel Definition

Name:	uddi-org:categorization:keyGenerator
Description:	Key Generator for UDDI Categorization tModels
UDDI Key (V3):	uddi:uddi.org:categorization:keygenerator
Derived V1,V2 format key:	uuid:f7343819-d5d6-3c02-aaeb-b0fe3c20dc5b
Categorization:	keyGenerator

11.5.2.4 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:categorization:keygenerator"
        xmlns ="urn:uddi-org:api_v3">
    <name>uddi-org:categorization:keyGenerator</name>
    <description>Key Generator for UDDI Categorization tModels</description>
    <overviewDoc>
        <overviewURL useType="text">
            http://uddi.org/pubs/uddi_v3.htm#categorizationKeyGen
        </overviewURL>
    </overviewDoc>
    <categoryBag>
        <keyedReference
            keyName="uddi-org:types:keyGenerator"
            keyValue="keyGenerator"
            tModelKey="uddi:uddi.org:categorization:types"/>
    </categoryBag>
</tModel>
```

11.5.3 Key Generator for UDDI Sort Order tModels

11.5.3.1 Introduction

This tModel establishes the UDDI sort order key partition. This key partition is used for the sort order tModels in this specification.

11.5.3.2 Design Goals

The design goal for the Key Generator for UDDI Sort Order tModels is to establish a key partition within a UDDI registry that allows UDDI to publish its canonical sort order tModels. All sort order tModels described in this document have keys derived in the key partition associated with this key generator.

11.5.3.3 tModel Definition

Name:	uddi-org:sortorder:keyGenerator

Description:	Key Generator for UDDI Sort Order tModels
UDDI Key (V3):	uddi:uddi.org:sortorder:keygenerator
Derived V1,V2 format key:	uuid: f6d9dd51-a5d6-37ae-8a56-08dd29be7572
Categorization:	keyGenerator

11.5.3.4 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:sortorder:keygenerator"
          xmlns ="urn:uddi-org:api_v3">
       <name>uddi-org:sortorder:keyGenerator</name>
       <description>Key Generator for UDDI Sort Order tModels</description>
       <overviewDoc>
            <overviewURL useType="text">
                http://uddi.org/pubs/uddi_v3.htm#sortorderKeyGen
            </overviewURL>
       </overviewDoc>
       <categoryBag>
            <keyedReference
                keyName="uddi-org:types:keyGenerator"
                keyValue="keyGenerator"
                tModelKey="uddi:uddi.org:categorization:types"/>
       </categoryBag>
</tModel>
```

11.5.4 Key Generator for UDDI Transport tModels

11.5.4.1 Introduction

This tModel establishes the UDDI transport key partition. This key partition is used for the transport tModels in this specification.

11.5.4.2 Design Goals

The design goal for the Key Generator for UDDI Transport tModels is to establish a key partition within a UDDI registry that allows UDDI to publish its canonical transport tModels. All transport tModels described in this document have keys derived in the key partition associated with this key generator.

11.5.4.3 tModel Definition

Name:	uddi-org:transport:keyGenerator
Description:	Key Generator for UDDI Transport tModels
UDDI Key (V3):	uddi:uddi.org:transport:keygenerator
Derived V1,V2 format key:	uuid:c356d69b-ec35-309a-a753-bdfd9fe67759
Categorization:	keyGenerator

11.5.4.4 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:transport:keygenerator"
          xmlns ="urn:uddi-org:api_v3">
      <name>uddi-org:transport:keyGenerator</name>
      <description>Key Generator for UDDI Transport tModels </description>
      <overviewDoc>
          <overviewURL useType="text">
              http://uddi.org/pubs/uddi_v3.htm#transportKeyGen
          </overviewURL>
      </overviewDoc>
      <categoryBag>
          <keyedReference
              keyName="uddi-org:types:keyGenerator"
              keyValue="keyGenerator"
              tModelKey="uddi:uddi.org:categorization:types"/>
      </categoryBag>
</tModel>
```

11.5.5 Key Generator for UDDI Protocol tModels

11.5.5.1 Introduction

This tModel establishes the UDDI protocol key partition. This key partition is used for the protocol tModels in this specification.

11.5.5.2 Design Goals

The design goal for the Key Generator for UDDI Protocol tModels is to establish a key partition within a UDDI registry that allows UDDI to publish its canonical protocol tModels. All protocol tModels described in this document have keys derived in the key partition associated with this key generator.

11.5.5.3 tModel Definition

Name:	uddi-org:protocol:keyGenerator
Description:	Key Generator for UDDI Protocol tModels
UDDI Key (V3):	uddi:uddi.org:protocol:keygenerator
Derived V1,V2 format key:	uuid:6ba0bd17-482b-3994-aeb9-d906aa2f80cb
Categorization:	keyGenerator

11.5.5.4 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:protocol:keygenerator"
          xmlns ="urn:uddi-org:api_v3">
      <name>uddi-org: protocol:keyGenerator</name>
      <description> Key Generator for UDDI Protocol tModels </description>
      <overviewDoc>
          <overviewURL useType="text">
              http://uddi.org/pubs/uddi_v3.htm#protocolKeyGen
          </overviewURL>
      </overviewDoc>
      <categoryBag>
          <keyedReference
```

```
                keyName="uddi-org:types:keyGenerator"
                keyValue="keyGenerator"
                tModelKey="uddi:uddi.org:categorization:types"/>
        </categoryBag>
</tModel>
```

11.5.6 UDDI Hosting Redirector Specification

11.5.6.1 Introduction

In UDDI, tModels are used to establish the existence of a variety of concepts and to point to their technical definitions. tModels of the specification sort are referenced by service bindings to describe the technical fingerprint of the business service. The hostingRedirector specification tModel establishes the concept of a hosting redirector Web service as described in Section 3.5.2.1 *accessPoint*. Refer to Appendix B *Using and Extending the useType Attribute* for usage details. Such Web services are used to dynamically obtain an access point for a Web service binding.

11.5.6.2 Design Goals

The design goal for the Hosting Redirector tModel is to facilitate recognition and discovery of Web service bindings that are associated with a hosting redirector service and to standardize the interface for such Web services.

11.5.6.3 tModel Definition

This tModel is used in bindingTemplate structures of Hosting Redirector Web services to indicate that the Web service implementation responds to the get_bindingDetail API with a bindingTemplate that contains an actual access point for the redirected Web service.

Name:	uddi-org:hostingRedirector
Description:	UDDI Hosting Redirector service specification
UDDI Key (V3):	uddi:uddi.org:specification:hostingredirector
Derived V1,V2 format key:	uuid:51c1535b-7e38-3860-9078-563d548420c5
Categorization:	specification

11.5.6.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel
 tModelKey="uddi:uddi.org:specification:hostingredirector">
  <name>uddi-org:hostingRedirector</name>
  <description>UDDI Hosting Redirector service specification</description>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#hostDir
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:specification"
      keyValue="specification"
      tModelKey="uddi:uddi.org:categorization:types"/>
   </categoryBag>
</tModel>
```

11.5.6.4 Example of Use

The following is an example of a bindingTemplate for a Hosting Redirector Web service. The bindingTemplate of a Web service making use of indirection via a hostingRedirector Web service contains the bindingKey of the hosting redirector service's bindingTemplate. The hosting redirector's bindingTemplate contains the accessPoint of the Hosting Redirector Web service as shown in the following example:

```
<bindingTemplate bindingKey="...." serviceKey="uddi:sk1....">
  <description xml:lang="en">My Redirector</description>
  <!-- URL of the Hosting Redirector service is in the
       accessPoint -->
  <accessPoint useType="hostingRedirector">
     ...the URL of the host redirection service...
  </accessPoint>
  <tModelInstanceDetails>
    <tModelInstanceInfo
      tModelKey="uddi:uddi.org:specification:hostingredirector"/>
  </tMobdelInstanceDetails>
</bindingTemplate>
```

11.5.7 UDDI Policy Description Specification

11.5.7.1 Introduction

In UDDI, tModels are used to establish the existence of a variety of concepts and to point to their technical definitions. tModels of the specification sort are referenced by service bindings to describe the technical fingerprint of the business service. The policy description specification tModel establishes the concept of a description of UDDI policy using schema-based XML.

11.5.7.2 Design Goals

The design goal for the Policy Description tModel is to enable discovery of UDDI policy for a UDDI node. UDDI policy is described in a schema driven XML document that is accessible from the access points of bindingTemplates that reside with UDDI node business entities.

11.5.7.3 tModel Definition

This tModel is used in bindingTemplate structures to indicate that the service implementation describes the policy of the node according to the policy description schema.

Name:	uddi-org:v3_policy
Description:	UDDI Policy Description service specification
UDDI Key (V3):	uddi:uddi.org:specification:v3_policy
Derived V1,V2 format key:	uuid:d52ce89c-01f8-3b53-a25e-89cfa5bbad17
Categorization:	specification

11.5.7.3.1 tModel Structure

This tModel is represented with the following structure:

```
<tModel
 tModelKey="uddi:uddi.org:specification:v3_policy">
  <name>uddi-org:v3_policy</name>
  <description>UDDI Policy Description service specification</description>
  <overviewDoc>
    <overviewURL useType="text">
         http://uddi.org/pubs/uddi_v3.htm#policyDesc
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:specification"
      keyValue="specification"
      tModelKey="uddi:uddi.org:categorization:types"/>
   </categoryBag>
</tModel>
```

11.5.7.4 Example of Use

The following is an example of a bindingTemplate for a Policy Description Web service. The policy description document is obtained using http:

```
<bindingTemplate bindingKey="...." serviceKey="uddi:sk1....">
  <accessPoint useType="endpoint">
     http://www.example.com/MyUDDIPolicy.xml
  </accessPoint>
  <tModelInstanceDetails>
    <tModelInstanceInfo
      tModelKey="uddi:uddi.org:specification:v3_policy"/>
  </tMobdelInstanceDetails>
</bindingTemplate>
```

12 Error Codes

The following list of error codes can be returned in the error code and error number (errCode and errno attributes) within a dispositionReport response to the API calls defined in this specification. The descriptions in this section are general and when used with the specific return information defined in the individual API call descriptions are useful for determining the reason for failures or other reasons. A list of the valid UDDI errCode (errno) values follows:

- **E_accountLimitExceeded**: (10160) Signifies that a save request exceeded the quantity limits for a given data type. Account limits are established based on the relationship between an individual publisher and an individual node. See your UDDI node's policy for account limits for details. Other nodes in the registry MAY NOT place additional restrictions on publishing limits established by a custodial node.
- **E_assertionNotFound**: (30000) Signifies that a particular publisher assertion cannot be identified in a save or delete operation.
- **E_authTokenExpired**: (10110) Signifies that the authentication token information has timed out.
- **E_authTokenRequired**: (10120) Signifies that an authentication token is missing or is invalid for an API call that requires authentication.
- **E_busy**: (10400) Signifies that the request cannot be processed at the current time.
- **E_fatalError**: (10500) Signifies that a serious technical error has occurred while processing the request.
- **E_historyDataNotAvailable**: (40010) Signifies that the requested history data is not available for the time period requested.
- **E_invalidCombination**: (40500) Signifies conflicting find qualifiers have been specified. The find qualifiers that caused the problem SHOULD be clearly indicated in the error text.
- **E_invalidCompletionStatus**: (30100) Signifies that one of the assertion status values passed is unrecognized. The completion status that caused the problem SHOULD be clearly indicated in the error text.
- **E_invalidKeyPassed**: (10210) Signifies that the uddiKey value passed did not match with any known key values. The details on the invalid key SHOULD be included in the dispositionReport element.
- **E_invalidProjection**: (20230) Signifies that an attempt was made to save a businessEntity containing a service projection where the serviceKey does not belong to the business designated by the businessKey. The serviceKey of at least one such businessService SHOULD be included in the dispositionReport.
- **E_invalidTime**: (40030) Signifies that the time period, the date/time, or the pair of date/time is invalid. The error structure signifies the condition that occurred and the error text clearly calls out the cause of the problem.
- **E_invalidValue**: (20200) This error code has multiple uses. This error code applies to the subscription APIs and the value set APIs. It can be used to indicate that a value that was passed in a keyValue attribute did not pass validation. This applies to checked value sets that are referenced using keyedReferences. The error text

SHOULD clearly indicate the key and value combination that failed validation. It can also be used to indicate that a chunkToken supplied is invalid. This applies in both subscription and value set APIs. The error text SHOULD clearly indicate the reason for failure.

- **E_keyUnavailable:** (40100) Signifies that the proposed key is in a partition that has already been assigned to some other publisher.
- **E_messageTooLarge**: (30110) Signifies that the message is too large. The upper limit SHOULD be clearly indicated in the error text.
- **E_noValuesAvailable**: (40200) Signifies that an attempt to retrieve valid values yielded no (additional) values.
- **E_requestDenied**: (20250) Signifies that a subscription cannot be renewed. The request has been denied due to either node or registry policy.
- **E_requestTimeout**: (20240) Signifies that the request could not be carried out because a needed Web service, such as validate_values, did not respond in a reasonable amount of time. Details identifying the failing Web service SHOULD be included in the dispositionReport element.
- **E_resultSetTooLarge**: (40300) Signifies that the UDDI node deems that a result set from an inquiry is too large, and requests to obtain the results are not honored, even using subsets. The inquiry that triggered this error should be refined and re-issued.
- **E_tokenAlreadyExists**: (40070) Signifies that one or more of the businessKey or tModelKey elements that identify entities to be transferred are not owned by the publisher identified by the *authInfo* element. The error text SHOULD clearly indicate which entity keys caused the error.
- **E_tooManyOptions**: (10030) DEPRECATED. Signifies that too many arguments were passed. The error text SHOULD clearly indicate the nature of the problem.
- **E_transferNotAllowed**: (40600) Signifies that the transfer of one or more entities has been by either the custodial node or the target node because the transfer token has expired or an attempt was made to transfer an unauthorized entity.
- **E_unacceptableSignature**: (40400) Indicates that the digital signature in the entity is missing or does not meet the requirements of the registry. The errInfo element provides additional details.
- **E_unrecognizedVersion**: (10040) Signifies that the value of the *namespace* attribute is unsupported by the *node* being queried.
- **E_unknownUser**: (10150) Signifies that the user ID and password pair passed in a get_authToken API is not known to the UDDI node or is not valid.
- **E_unsupported**: (10050) Signifies that the implementer does not support a feature or API.
- **E_userMismatch**: (10140) Signifies that an attempt was made to use the publishing API to change data that is controlled by another party.
- **E_valueNotAllowed**: (20210) Signifies that a value did not pass validation because of contextual issues. The value may be valid in some contexts, but not in the context used. The error text MAY contain information about the contextual problem.
- **E_unvalidatable**: (20220) Signifies that an attempt was made to reference a value set in a keyedReference whose tModel is categorized with the unvalidatable categorization.

Non-error conditions are not reported by way of SOAP Faults but are instead reported using an empty response message.

12.1 Common Error Conditions

In Chapter 5 *UDDI Programmers APIs,* a list of applicable error conditions are described in the Caveats sections of each API description. In addition to these error conditions that occur within each API's context, there are error conditions that can occur at any time, for any of the UDDI APIs, for example, due to unavailability of the UDDI node or heavy network traffic. Unless there are special considerations associated with a given API, these overarching error conditions are not listed in the Caveats sections of the APIs. The error conditions that apply to the entire UDDI API are:

- E_authTokenExpired
- E_authTokenRequired
- E_busy
- E_fatalError
- E_requestTimeout
- E_unrecognizedVersion
- E_unsupported.

13 Related Standards and Specifications

This document refers to other UDDI documents as well as several widely recognized standards and specifications. Detailed information on these may be found as follows:

13.1 UDDI Specifications and documents

UDDI Version 2.0 Operator's Specification

http://uddi.org/pubs/operators_v2.pdf

UDDI Version 2.0 Custody Transfer Schema

http://uddi.org/schema/uddi_2_custody.xsd

UDDI Version 2.0 Programmer's API Specification

http://uddi.org/pubs/ProgrammersAPI_v2.pdf

UDDI Version 2.0 API Schema

http://uddi.org/schema/uddi_v2.xsd

UDDI Version 2.0 Replication Specification

http://uddi.org/pubs/replication_v2.pdf

UDDI Version 2.0 Replication Schema

http://uddi.org/schema/uddi_v2_replication.xsd

13.2 Standards and other Specifications

Specification	Location
Augmented BNF for Syntax Specifications: ABNF	http://www.ietf.org/rfc/rfc2234
Codes for the Representation of Names of Languages–Part 2: Alpha-3 Code	http://www.loc.gov/standards/iso639-2/langhome.html
Extensible Markup Language (XML) 1.0	http://www.w3.org/TR/1998/REC-xml-19980210.html
HTML 3.2 Reference Specification	http://www.w3.org/TR/REC-html32
HTTP Over TLS	http://www.ietf.org/rfc/rfc2818
Hypertext Transfer Protocol -- HTTP/1.1	http://www.w3.org/Protocols/HTTP/1.1/rfc2616.pdf
Internet Security Glossary	http://www.ietf.org/rfc/rfc2828
Internet X.509 Public Key Infrastructure	http://www.ietf.org/rfc/rfc2459
ISO 3166-1: The Code List	http://www.din.de/gremien/nas/nabd/iso3166ma/codlstp1/index.html
ISO/IEC9075-2:1999(E) Database Language - SQL	http://www.cse.iitb.ernet.in:8000/proxy/db/~dbms/Data/Papers-Other/SQL1999/
Multipurpose Internet Mail Extensions (MIME) Part One	http://www.ietf.org/rfc/rfc2045

Specification	Location
Schema Centric XML Canonicalization	http://uddi.org/pubs/SchemaCentricCanonicalization.htm
Simple Object Access Protocol (SOAP) 1.1	http://www.w3.org/TR/SOAP
SMTP - Simple Mail Transfer Protocol	http://www.ietf.org/rfc/rfc2821
SSL 3.0 Specification	http://home.netscape.com/eng/ssl3/index.html
Tags for the Identification of Languages	http://www.ietf.org/rfc/rfc3066
Terminology for Policy-Based Management	http://www.ietf.org/rfc/rfc3198
The Unicode Standard	http://www.unicode.org/unicode/standard/standard.html
Unicode Standard Annex #15 UNICODE NORMALIZATION FORMS	http://www.unicode.org/unicode/reports/tr15/
Unicode Technical Standard #10 UNICODE COLLATION ALGORITHM	http://www.unicode.org/unicode/reports/tr10/
Uniform Resource Identifiers (URI): Generic Syntax	http://www.ietf.org/rfc/rfc2396
URLs for Telephone Calls	http://www.ietf.org/rfc/rfc2806
UTF-16, an encoding of ISO 10646	http://www.ietf.org/rfc/rfc2781
UTF-8, a transformation format of ISO 10646	http://www.ietf.org/rfc/rfc2279
UUIDs and GUIDs	http://uddi.org/pubs/draft-leach-uuids-guids-01.txt
Web Services Description Language (WSDL) 1.1	http://www.w3.org/TR/wsdl
XML Schema	http://www.w3.org/XML/Schema#dev

A Appendix A: Relationships and Publisher Assertions

UDDI includes a relationship feature based on "publisher assertions". Publisher assertions are the basis for a mechanism to allow registered businessEntity elements to be linked in a manner that conveys a specific type of relationship. Publisher assertions are used to establish visible, reciprocal relationships between businessEntity elements in a way that once completed, a matching set of assertions can be seen by the find_relatedBusinesses and find_business calls.

In order to make it possible for either party in a relationship to have some control over the visibility of the relationship, relationships are only visible when both parties of a potential relationship agree that such a relationship exists at a given point in time. This is done by both parties publishing publisherAssertions that are identical, with the exception of any optional digitalSignature elements that are assigned to the publisherAssertions. This addresses a problematic scenario that arises when one party falsely claims that it is related to some other party.

Only publishers that control one of the businesses involved in the relationship are allowed to assert that a relationship exists. When a publisher controls both businessEntity structures involved in the relationship, a single publisherAssertion element satisfies the relationship (e.g., a second assertion is not required to form the relationship). In the case where a different publisher controls each businessEntity involved in such an expression, both parties must assert identical information about a specific relationship before UDDI surfaces any information about the relationship. In cases where two parties are involved and both parties do not agree as to the details of a given assertion, there is no requirement for either party to complete an assertion. No relationship is exposed via the Inquiry API in this case.

A.1 Example

The following picture shows the start of an assertion process:

Joe and Xina each manage a businessEntity within UDDI. As we start our scenario, both Joe and Xina have registered a businessEntity in the same UDDI registry. Joe and Xina wish to make it possible for users of this UDDI registry to find out that the two businesses are related and are in fact part of the same business, with Business1 being a parent-business.

To make the relationship visible for anyone who calls find_relatedBusinesses passing either businessKey as a starting point, Joe and Xina must both make publisherAsssertion. As the name of the element suggests, a publisherAssertion is an assertion made by a publisher who is expressing a particular fact about a business registration and its relationship to some other business data within UDDI.

Joe uses an add_publisherAssertions SOAP message to make a new publisherAssertion about Business1 and Business 2, expressing the fact that Business 1 is a corporate holding company. This message looks like:

```
<add_publisherAssertions xmlns="urn:uddi-org:api_v3" >
   <authInfo>FFFFF</authInfo>
   <publisherAssertion>
        <fromKey>BK1</fromKey>
        <toKey>BK2</toKey>
        <keyedReference
            tModelKey="uddi:uddi.org:relationships"
            keyName="Holding Company"
            keyValue="parent-child" />
   </publisherAssertion>
</add_publisherAssertions>
```

In this example, we see that Joe asserts that the businessEntity with the businessKey value of BK1 is the parent holding company of the businessEntity with the businessKey value of BK2. The publisherAssertion could be signed with an XML digital signature if Joe feels this is important.

Because only Joe has asserted this fact, the information about the relationship is not yet visible via the inquiry API call, find_relatedBusinesses. Joe knows that for this assertion to become visible, the publisher of the businessEntity that has the businessKey BK2 must also express the same assertion. Joe calls Xina to let her know he wants her to make the assertion.

In order to see the data that she must express, Xina sends a get_assertionStatusReport to her UDDI node. From the resulting assertionStatusReport, Xina sees that there is indeed an unmatched assertion listed against her businessEntity Business 2. Since Joe has contacted her and she agrees that the relationship should be visible within UDDI, Xina sends the exact same assertion (with a different authInfo credential) to her UDDI node. The publisherAssertion can be signed with an XML digital signature if such a signature is desired by Xina.

The UDDI registry now sees both assertions made by the two publishers, each of whom control one of the two businesses involved. After checking that the requesting parties each control half of the relationship, UDDI matches the assertions together and the status of the relationship becomes complete.

After this is done, anyone who calls the Inquiry API call, find_relatedBusinesses, and passes either BK1 or BK2 as the businessKey value will see the relationship. Prior to both publishers asserting this same fact, the data about the relationship is not visible via the Inquiry API.

A.2 Managing relationship visibility

The UDDI Publish API defines several APIs to allow assertions to be managed by UDDI publishers. These APIs fall into two general categories: administrative helpers and maintenance functions. The administrative helpers allow the publisher to see assertions that their businesses are involved in. In particular, the get_assertionStatusReport is the most useful for determining whether any assertions involving the business registrations owned by a publisher are incomplete. This provides information pertaining to assertions the publisher is expecting and also presents information about other publishers' attempts at making unexpected assertions.

The maintenance functions allow publishers to deal with all assertions as a single group (e.g. get_publisherAssertions / set_publisherAssertions) or individually (e.g. add_publisherAssertions / delete_publisherAssertions). The set_publisherAssertions call should be used with caution as it replaces all of the publisherAssertions for a publisher and can be used to invalidate existing completed relationships. The latter APIs are useful for adding one publisherAssertion at a time without having to keep track of all previous assertions.

B Appendix B: Using and Extending the useType Attribute

UDDI provides type information through the useType attribute on the following UDDI elements: accessPoint, overviewURL, discoveryURL, contact, address, email and phone. The useType attribute is intended to provide information on how to use or invoke the resource contained within the element. This Appendix establishes and explains common values and conventions for the useType attribute in the context of certain elements, as well as a model for establishing new common values and conventions.

B.1 accessPoint

Four common values for providing type information about the accessPoint are: "endPoint", "wsdlDeployment", "bindingTemplate", and "hostingRedirector.

B.1.1 Using the "endPoint" value

Typically, the network address of a Web service described by a bindingTemplate is found in the accessPoint element. This common behavior is denoted by using the string "endpoint" as the value of the accessPoint. Decorating an accessPoint with a useType="endPoint" signifies that a user or application can invoke a Web service at that address. A sample of such behavior is as follows:

```
<bindingTemplate bindingKey="uddi:example.org:catalog">
   <description xml:lang="en">
        Browse catalog Web service
   </description>
   <accessPoint useType="endPoint">
        http://www.example.org/CatalogWebService
   </accessPoint>
   <tModelInstanceDetails>
        <tModelInstanceInfo tModelKey="uddi:example.org:catalog_interface"/>
        <tModelInstanceInfo tModelKey="uddi:uddi.org:transport:http"/>
   </tModelInstanceDetails>
</bindingTemplate>
```

In the example above, a client would be able to parse the bindingTemplate and discover the end point of the Web service itself. However, the information about how to invoke that Web service would be modeled using tModels. All interface information about that service would be represented by the tModelInstanceInfo structures contained as children of the bindingTemplate.

The client knows the transport of the accessPoint either by checking to see if a protocol tModel has been associated with the bindingTemplate or inspecting the URI prefix. In the example above, the http transport was used, denoted by the tModelInstanceInfo.

UDDI RECOMMENDS that endpoints for phone, fax and modem communication follow the guidelines outlined in RFC 2806 *URLs for Telephone Calls*[51]. Following such a convention for a phone number accessPoint would result in the following bindingTemplate sample:

```
<bindingTemplate bindingKey="uddi:example.org:catalog">
   <description xml:lang="en">
        Browse catalog Web service
   </description>
   <accessPoint useType="endPoint">
        tel:+1-512-555-1212
   </accessPoint>
   <tModelInstanceDetails>
        <tModelInstanceInfo tModelKey="uddi:uddi.org:transport:telephone"/>
   </tModelInstanceDetails>
</bindingTemplate>
```

B.1.2 Using the "wsdlDeployment" value

Instead of directly providing the network address in the accessPoint, it is occasionally useful or necessary to provide this information through indirect means. One common scenario for such a behavior is when the accessPoint is embedded within a WSDL file. In such a scenario, the UDDI accessPoint contains the address of the WSDL file, and the client then must retrieve the WSDL file and extract the end point address from the WSDL file itself.

In this case, decorating the UDDI accessPoint with a useType="wsdlDeployment" is appropriate. A sample of such behavior is as follows:

```
<bindingTemplate bindingKey="uddi:example.org:catalog">
   <description xml:lang="en">
        Browse catalog Web service
   </description>
   <accessPoint useType="wsdlDeployment">
        http://www.example.org/CatalogWebService/catalog.wsdl
   </accessPoint>
   <categoryBag>
       <keyedReference keyName="uddi-org:types:wsdl"
       keyValue="wsdlDeployment"
       tModelKey="uddi:uddi.org:categorization:types"/>
   </categoryBag>
</bindingTemplate>
```

In the example above, a client would be able to parse the result of the bindingTemplate and determine the end point of the Web service within the WSDL file discovered in the accessPoint element. Note that the bindingTemplate has also been categorized with the "wsdlDeployment" value from the uddi.org:categorization:types scheme so that it can be discovered through a find_binding API call.

B.1.3 Using the "bindingTemplate" value

Another form of indirection uses UDDI itself to discover the location of the final end point. Categorizing a bindingTemplate with either "bindingTemplate" or "hostingRedirector" specifies that the accessPoint contains a bindingKey intended to be used in a get_bindingDetail API call to a UDDI registry Inquiry API Set. How the resultant bindingTemplate is interpreted depends on which one of the two methods described below is used. In the case of "bindingTemplate", a bindingKey refers to another binding within the same UDDI registry.

[51] http://www.ietf.org/rfc/rfc2806

For example, suppose tempuri.com, the well known but fictitious maker of crispy batter coating for fried foods, contracts with the equally fictitious Web service hosting company ws-o-rama.com to host tempuri's Web service that exposes its product catalog. Tempuri.com wishes to publish a bindingTemplate for this service in the UDDI Business Registry, but wishes to leave the details of the technical implementation and its description up to ws-o-rama.com who may wish to change them over time. To do this, tempuri.com and ws-o-rama.com use bindingTemplate indirection. In the UDDI Business Registry the bindingTemplate in tempuri's businessEntity that describes this service appears as follows:

```
<bindingTemplate bindingKey="uddi:tempuri.com:catalog">
   <description xml:lang="en">
        Browse catalog Web service
   </description>
   <accessPoint useType="bindingTemplate">
        uddi:ws-o-rama.com:tempuri:bt1
   </accessPoint>
</bindingTemplate>
```

Here, the bindingTemplate describing tempuri's catalog browsing Web service uses an accessPoint to refer to the bindingTemplate (also in the UDDI Business Registry) whose bindingKey is uddi:ws-o-rama.com:tempura:bt1. When a client does a get_bindingDetail asking for the bindingTemplate with that key, the following bindingTemplate is returned:

```
<bindingTemplate bindingKey="uddi:ws-o-rama.com:tempuri:bt1">
   <description xml:lang="en">
        Tempuri.com's catalog browsing service hosted by ws-o-rama
   </description>
   <accessPoint useType="endPoint">
        http://sf1.ws-o-rama.com/tempuri/catalog/
   </accessPoint>
   <tModelInstanceDetails>
        <tModelInstanceInfo tModelKey="uddi:..."/>
   </tModelInstanceDetails>
</bindingTemplate>
```

This bindingTemplate describes the actual Web service that is to be used.

B.1.4 Using the "hostingRedirector" value

It may be necessary to "hide" the network address of a Web service from unauthorized access. In this case, a useType="hostingRedirector" can be used to indicate that the client must follow a longer path of indirection to retrieve the accessPoint. In the bindingTemplate, the client will discover a bindingKey, which will lead to a bindingTemplate that contains the end point for a Web service which supports the UDDI get_bindingDetail Web Service. The client will then be able to re-issue the get_bindingDetail message with the original key, presumably with authentication, to this other Web service. Such an indirection mechanism allows a Web service to be discoverable but not accessible from a given node.

For example, tempuri.com uses ws-o-rama.com to host more than just its publicly visible catalog browsing service. In particular it has a number of services it does not wish to expose fully in the UDDI Business Registry. Instead, it wishes to keep their full definition in its private UDDI registry, which ws-o-rama.com also happens to host, and supply the end points to these Web services only to authorized inquirers.

In particular, tempuri has a Web service that its suppliers use to bill it for goods they deliver. In the UDDI Business Registry, tempuri publishes the following bindingTemplate, which contains a bindingKey.

```
<bindingTemplate bindingKey="uddi:tempuri.com:billing">
   <description xml:lang="en">
        Tempuri supplier billing Web service
   </description>
   <accessPoint useType="hostingRedirector">
        uddi:ws-o-rama.com:tempuri:bt47
```

```
   </accessPoint>
</bindingTemplate>
```

Here, the bindingTemplate describing tempuri's supplier billing Web service uses an accessPoint to refer to the bindingTemplate (also in the UDDI Business Registry) whose bindingKey is uddi:ws-o-rama.com: tempuri:bt47. Note that the useType equals "hostingRedirector" which indicates that the bindingKey refers to a hostingRedirector service. When a client does a get_bindingDetail (on the UDDI Business Registry) asking for the bindingTemplate with that key, the following indirect bindingTemplate is returned:

```
<bindingTemplate bindingKey="uddi:ws-o-rama.com:tempura:bt47">
   <description xml:lang="en">
        Hosting Redirector Service for Tempuri.com
        hosted by ws-o-rama.com
   </description>
   <accessPoint useType="endPoint">
        http://sf1.ws-o-rama.com/tempuri/uddi/inquiry
   </accessPoint>
   <tModelInstanceDetails>
        <tModelInstanceInfo
            tModelKey="uddi:uddi.org:specification:hostingredirector"/>
   </tModelInstanceDetails>
</bindingTemplate>
```

This bindingTemplate describes the hosting redirector Web service hosted for tempuri.com by ws-o-rama.com. By definition, it responds to the get_bindingDetail API call using SOAP over HTTP. The description in the bindingTemplate says it responds only to authorized requests. For authorized clients, invoking get_bindingDetail, passing the key of the original bindingTemplate (uddi:tempuri.com:billing) retrieves the following bindingTemplate:

```
<bindingTemplate bindingKey="uddi:tempuri.com:billing">
   <description xml:lang="en">
        Tempuri.com's supplier billing browsing service
   hosted by ws-o-rama.com
   </description>
   <accessPoint useType="endPoint">
        http:sf1.ws-o-rama.com/tempuri/billing/
   </accessPoint>
   <tModelInstanceDetails>
        <tModelInstanceInfo tModelKey="uddi:..."/>
   </tModelInstanceDetails>
</bindingTemplate>
```

This bindingTemplate describes the actual Web service that is to be used.

B.2 overviewURL

The overviewURL appears as a child of the overviewDoc, which appears twice in the UDDI information model, once with tModel element and once with tModelInstanceInfo element. There are two conventions established, "text" and "wsdlInterface".

B.2.1 Using the "text" value

Using the useType of "text" signifies that textual information meant for human consumption is available at the resource specified by that URL. It is appropriate both for tModel and tModelInstanceInfo elements.

The following example demonstrates an overviewURL that points to a .pdf file that discusses more about the implementation of the **uddi:tempuri.org:catalog_interface** tModel implemented. It might explain particular details about this implementation that a developer needs to know.

```
<tModelInstanceInfo tModelKey="uddi:tempuri.org:catalog_interface">
   <instanceDetails>
        <overviewDoc>
            <description xml:lang="en">This overviewURL provides additional
information about this Web service.</description>
            <overviewURL useType="text">http://tempuri.org/info.pdf</overviewURL>
        </overviewDoc>
   </instanceDetails>
</tModelInstanceInfo>
```

B.2.2 Using the "wsdlInterface" value

The "wsdlInterface" value signifies that a WSDL file is located at this resource. Such a WSDL file has no implementation information, but exists purely as an abstract interface document. Using this convention within a tModelInstanceInfo is not appropriate.

For example, the tModel below has two overviewDocs. The first, which uses the "wsdlInterface" value, denotes that the overviewURL points to WSDL file. The second overviewDoc provides additional information about the tModel, using the "text" value.

```
<tModel tModelKey="uddi:tempuri.org:catalog_interface">
  <name>urn:tempuri.org:catalog_interface</name>
  <description>This WSDL interface has a set of APIs for querying the catalog
service.</description>
  <overviewDoc>
    <overviewURL useType="wsdlInterface">
       http://www.tempuri.org/wsdl/catalog_interface.wsdl
    </overviewURL>
  </overviewDoc>
  <overviewDoc>
    <overviewURL useType="text">
         http://www.tempuri.org/wsdl/interface_info.pdf
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:wsdl"
      keyValue="wsdlSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:soap"
      keyValue="soapSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:xml"
      keyValue="xmlSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:specification"
      keyValue="specification"
      tModelKey="uddi:uddi.org:categorization:types"/>
   </categoryBag>
</tModel>
```

B.3 discoveryURL

The discoveryURL only appears at the businessEntity level. Two conventions have been established: "businessEntity" and "homepage".

B.3.1 Using the "businessEntity" value

Per the UDDI Specification, discoveryURLs qualified with the value "businessEntity" point to XML documents of the type businessEntity, as defined in the UDDI API Schema.

```
<discoveryURL useType="businessEntity">
    http://www.example.com?businessKey= uddi:example.com:registry:sales:53
</discoveryURL>
```

B.3.2 Using the "homepage" value

Frequently, businesses and providers want to register HTTP-accessible HTML Web "homepage" information. The discoveryURL useType of "homepage" satisfies that purpose:

```
<discoveryURL useType="homepage">
    http://www.example.com
</discoveryURL>
```

B.4 Contact

No conventions have been established.

B.5 Address

No conventions have been established.

B.6 Phone

No conventions have been established.

B.7 Email

No conventions have been established.

B.8 Designating a new useType value

While the useType conventions listed above cover a set of common cases, there may be situations which a new useType attribute for an element needs to be designated. UDDI RECOMMENDS[52] that the creation of new useType values should map to UDDI tModelKeys. In such a way, a client can look up the tModel to understand the semantic meaning of this unknown useType attribute.

For example, imagine a new kind of indirection mechanism for the accessPoint element is created that involves a new protocol. In this example the new protocol refers to files that have a file extension of .nwp (for NeW Protocol). In order to decorate accessPoints with a useType attribute that refers to this protocol, a new tModel should be published to UDDI, categorized by the UDDI Types taxonomy, as follows:

[52] Because the datatype of the useType is xsd:string, any value can be placed in this attribute, such as a well known URI. However, by following the recommended practice, there is a known way of determining more information about the meaning of the alien useType value.

```
<tModel tModelKey="uddi:tempuri.org:tmodel:newprotocol">
  <name>tempuri.org:tModel:NewProtocol</name>
  <description>A tModel that represents the useType for the new protocol
  </description>
  <overviewDoc>
    <overviewURL useType="text">
         http://tempuri.org/NewProtocol/about.htm
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:useTypeDesignator"
      keyValue="useTypeDesignator"
      tModelKey="uddi:uddi.org:categorization:types"/>
   </categoryBag>
</tModel>
```

This tModel establishes the new useType value "uddi:tempuri.org:tmodel:newprotocol". Its categorization as a "useTypeDesignator" semantically signifies this purpose. When using this useType in an accessPoint, the key of the tModel is used as the value, as follows:

```
<bindingTemplate bindingKey="uddi:tempuri.com:some_service">
   <description xml:lang="en">
        Tempuri Web service
   </description>
   <accessPoint useType="uddi:tempuri.org:tmodel:newprotocol">
        http://www.tempuri.org/service.nwp
   </accessPoint>
</bindingTemplate>
```

When a client encounters this unfamiliar useType in a bindingTemplate, the client can issue a get_tModelDetail with the value found in the useType attribute, thus learning the technical nature of this accessPoint.

C Appendix C: Supporting Subscribers

This appendix describes scenarios where subscription is useful and provides examples of how to use subscription.

C.1 Subscription Scenarios

Subscription support is intended to provide clients or subscribers with the ability to register interest in receiving information concerning changes made to the specified subsets of the data in a UDDI Registry and then to obtain updates to the subset on a periodic basis as the data changes. There are several scenarios for which subscription is useful. Among the common uses are:

- **Notification of new businesses or services that register** - A subscriber may be interested in receiving notification whenever a new business or service becomes available that conforms to a pre-defined set of criteria. This might be a business that could be a new potential business partner. The subscriber can simply register to be informed when new users of certain tModels emerge, for instance, expanding his potential base of suppliers.
- **Monitoring of existing businesses or services** - A subscriber may be interested in receiving an update on a particular business or service whenever it is altered in any manner, including deletion. A subscriber can use subscription to monitor a particular service or business, receiving a notification whenever there is a change in the registry entry for that particular service or business.
- **Obtaining Registry data for use in a private UDDI Registry** – A subscriber might represent a private UDDI registry installation operated by a particular industry. A node might, for example, wish to augment its own content with relevant entries from the UDDI Business registry. Subscription allows them to track and receive all changes of interest for integration into their own registry. In another example, the private registry might even wish to mirror all of the data present in another registry with which it does not replicate. This capability is by creating a small number of subscriptions.
- **Obtaining Registry data for use by a Registrar** – A subscriber might be a Registrar who facilitates creation and maintenance of Registry entries in the UDDI Business Registry for a specialized set of customers. As it happens, information about existing relationships in the Registry which various business partners maintain with each other affect the maintenance of business relationships with these customers. The Registrar offers a service to these customers to automatically maintain their relationship information based on a profile these customers maintain with the Registrar. Subscription enables the Registrar to perform this service by providing notification of changes in business relationships
- **Obtaining Registry data for use by eMarketplaces** – A subscriber might represent an eMarketplace which has become a portal for a particular industry. The eMarketplace combines and organizes all of the Web service offerings across multiple UDDI Registries which interest its customers, offering them one-stop shopping. Subscription helps this eMarketplace keep abreast of new and changing service offerings without having to troll these Registries on a continuous basis.

C.2 Using Subscription

This section describes the steps involved in using the subscription API set and provides several examples.

C.2.1 Steps for Creating a Subscription

Establishing a subscription and monitoring the results is in general a multi-step process:

1. Create a service to receive the notifications the registry provides. Subscribers can choose to receive subscriptions through either HTTP/SOAP Web services they implement, or via e-mail. As many bindingTemplates of either type can be defined as desired if multiple endpoints are required, but only one bindingTemplate can be associated with a given subscription.
2. Register the service with the registry, which the node is to use to deliver notifications. This requires registering of a single bindingTemplate describing the service. Note that a single service can be used to receive the notifications for multiple subscriptions or, if the subscriber chooses, separate services can be created for each subscription.
3. Select the filter criteria to be used for the subscription. This requires some care – choosing more restrictive criteria reduces the result set returned making it simpler to analyze. In general, subscribers should insure that the subscriptionFilter criterion they use is as restrictive as possible.
4. Save the subscription request using save_subscription.
5. Process the incoming HTTP/SOAP or e-mail notifications as desired.

C.2.2 Subscription Examples

In most cases, the actual values of the various keys and authInfo elements involved in these examples are not shown, but are instead represented in ***bold italics*** descriptions.

1. Create a subscription to track changes in businesses which offer services in the Motor Vehicle Parts industry. To accomplish this, the subscriber needs to register a service with a bindingTemplate indicating a desire to receive notifications through an endpoint which corresponds to either a notify_subscriptionListener service (which the subscriber implements), or via email. Here are examples of each of these two types of bindings:

```
<save_binding xmlns="urn:uddi-org:api_v3">
     <authInfo>myAuthCode</authInfo>
     <bindingTemplate  bindingKey="" serviceKey="uddi:myservicekey">
         <description>notify_subscriptionListener binding for
             my subscription.
         </description>
         <accessPoint URLType="https">
             https://www.myCompany.com/services/notify_subscriptionListener
         </accessPoint>
         <tModelInstanceDetails>
             <tModelInstanceInfo
                 tModelKey="uddi:uddi.org:v3_subscriptionlistener" />
         </tModelInstanceDetails>
     </bindingTemplate>
</save_binding>
<save_binding xmlns="urn:uddi-org:api_v3">
     <authInfo>myAuthCode</authInfo>
     <bindingTemplate  bindingKey="" serviceKey="uddi:myservicekey">
         <description>E-mail binding for my subscription.</description>
         <accessPoint URLType="email">
             mailto:mySubscriberEmail@xyz.com
         </accessPoint>
     </bindingTemplate>
</save_binding>
```

Let's use the notify_subscriptionListener binding and register a subscription request to return a notification every 5 days. Our subscriptionFilter will limit the results to new, changed or deleted services categorized as "Motor Vehicle Supplies and New Parts Wholesalers" using the North American Industry Classification System (NAICS). Note that the brief attribute is used to force only entity keys to be returned in the results:

```
<save_subscription xmlns="urn:uddi-org:sub_v3">
     <authInfo>myAuthCode</authInfo>
     <subscriptions>
         <subscription brief="true">
             <subscriptionFilter>
                 <find_service xmlns="urn:uddi-org:api_v3" >
                     <findQualifiers>
                         <findQualifier>
                             uddi:uddi.org:findqualifier:sql99:like
                         </findQualifier>
                     </findQualifiers>
                     <categoryBag>
                         <keyedReference
                             tModeKey="uddi:uddi.org:ubr:taxonomy:naics"
                             keyName="Motor Vehicle Parts"
                             keyValue="42112_"/>
                     </categoryBag>
                 </find_service>
             </subscriptionFilter>
                 <bindingKey>
                     bindingKeyOfTheClientsNotifySubscriptionListenerService
                 </bindingKey>
                 <notificationInterval>P5D</notificationInterval>
                 <maxEntities>1000</maxEntities>
         </subscription>
     </subscriptions>
</save_subscription>
```

An example of the node's subsequent invocation of the client implemented notify_subscriptionListener API is shown below. The call to notify_subscriptionListener by the node transmits data for January 2002 using the lexical representation of dateTime as defined by [ISO 8601], in the extended format of CCYYMMDDThh:mm:ss where "CC" represents the century, "YY" the year, "MM" the month and "DD" the day. The letter "T" is the date/time separator and "hh", "mm", "ss" represent hour, minute and second respectively:

```
<notify_subscriptionListener>
    <subscriptionResultsList>
        <coveragePeriod>
            <startPoint>20020101T00:00:00</startPoint>
            <endPoint>20020131T00:00:00</endPoint>
        </coveragePeriod>
        <subscription brief="true">
            <subscriptionFilter>
                <find_service xmlns="urn:uddi-org:api_v3" >
                    <findQualifiers>
                        <findQualifier>
                            uddi:uddi.org:findqualifier:sql99:like
                        </findQualifier>
                    </findQualifiers>
                    <categoryBag>
                        <keyedReference
                            tModeKey="uddi:uddi.org:ubr:taxonomy:naics"
                            keyName="Motor Vehicle Parts"
                            keyValue="42112_"/>
                    </categoryBag>
                </find_service>
                </subscriptionFilter>
                <bindingKey>
                    bindingKeyOfTheClientsNotifySubscriptionListenerService
                </bindingKey>
                <notificationInterval>P5D</notificationInterval>
                <maxEntities>1000</maxEntities>
                <expiresAfter>20030101T00:00:00</expiresAfter>
        </subscription>
        <keyBag>
            <deleted>false</deleted>
            <serviceKey>matchingKey1</serviceKey>
            <serviceKey>matchingKey2</serviceKey>
            <serviceKey>matchingKey3</serviceKey>
            <serviceKey>matchingKey4</serviceKey>
        </keyBag>
        <keyBag>
            <deleted>true</deleted>
            <serviceKey>matchingKey5</serviceKey>
            <serviceKey>matchingKey6</serviceKey>
        </keyBag>
    </subscriptionResultsList>
</notify_subscriptionListener>
```

2. Use get_subscriptionResults to retrieve chunked subscription results synchronously. In this example, we reuse the same <save_subscription> API call shown in example (1) with the exception that no <bindingKey> is provided. This prevents notifications from being sent under the assumption that the client desires to use the get_subscriptionResults API to do this instead. Here is a typical use of the get_subscriptionResults API. The request shown retrieves results available for changes made since the beginning of January 2002:

```
<get_subscriptionResults>
     <authInfo>myAuthCode</authInfo>
     <subscriptionKey>mySubscriptionKey</subscriptionKey>
     <coveragePeriod>
         <startPoint>20020101T00:00:00</startPoint>
     </coveragePeriod>
</get_subscriptionResults>
```

The above request should return all subscription results matching the criterion we saved in the subscription for entities whose last date of change was on or after January 1st, 2002, up through the present time. A subscriptionResultsList is returned synchronously by the get_subscriptionResults API:

```
<subscriptionResultsList>
     <chunkToken>"nodeGeneratedToken"</chunkToken>
     <coveragePeriod>
         <startPoint>20020101T00:00:00</startPoint>
     </coveragePeriod>
     <subscription  brief="true">
         <subscriptionFilter>
             <find_service xmlns="urn:uddi-org:api_v3" >
                 <findQualifiers>
                     <findQualifier>
                         uddi:uddi.org:findqualifier:sql99:like
                     </findQualifier>
                 </findQualifiers>
                 <categoryBag>
                     <keyedReference
                         tModeKey="uddi:uddi.org:ubr:taxonomy:naics"
                         keyName="Motor Vehicle Parts"
                         keyValue="42112_"/>
                 </categoryBag>
             </find_service>
         </subscriptionFilter>
         <bindingKey>
             bindingKeyOfTheClientsNotifySubscriptionListenerService
         </bindingKey>
         <notificationInterval>P5D</notificationInterval>
         <maxEntities>1000</maxEntities>
         <expiresAfter>20030101T00:00:00</expiresAfter>
     </subscription>
     <keyBag>
         <deleted>false</deleted>
         <serviceKey>matchingKey1</serviceKey>
         <serviceKey>matchingKey2</serviceKey>
         <serviceKey>matchingKey3</serviceKey>
         <serviceKey>matchingKey4</serviceKey>
     </keyBag>
</subscriptionResultsList>
```

Unfortunately, there are too many results for us to get them in one group since we noticed that the chunkToken value of "***nodeGeneratedToken***" returned was not "0", so chunking is being used. We use this "***nodeGeneratedToken***" to call get_subscriptionResults again to retrieve the next group of results:

```
<get_subscriptionResults>
     <authInfo>myAuthCode</authInfo>
     <subscriptionKey>mySubscriptionKey</subscriptionKey>
     <coveragePeriod>
         <startPoint>20020101T00:00:00</startPoint>
     </coveragePeriod>
     <chunkToken>nodeGeneratedToken</chunkToken>
</get_subscriptionResults>
```

The remaining results are returned synchronously in a subscriptionResultsList structure as before. The last results have been returned when the chunkToken returned in the subscriptionResultsList is "0".

D Appendix D: Internationalization

As part of the aim of providing a registry for *universal* description, discovery and integration of business entities and their services, the UDDI registry design includes support for internationalization features. Most of these internationalization features are directly exposed to end users through the API sets. Others are built into the design in order to enable the use of the UDDI registry as an international Web services discovery and description mechanism with multilingual descriptions of business entities worldwide. This appendix provides examples for some of the internationalization features supported:

- multilingual descriptions, names and addresses
- multiple names in the same language
- internationalized address format
- language-dependent collation

D.1 Multilingual descriptions, names and addresses

The description, name, personName, or address elements may each have an xml:lang attribute to indicate the language used in the content of these elements. Thus names and addresses, for example, may have characters from language scripts other than the Latin script found in ASCII. Similarly, variants of names, due to transliteration, e.g. romanization, to different languages, are indicated through the use of the xml:lang attribute. The rules and syntax governing the xml:lang data type are as defined in Section 3.3.2.3 *name*.

The following shows an example of romanization where the primary name of the business (a Chinese flower shop) is in Chinese, and its alternative name is a romanization:

```
<businessEntity . . . >
  ........
  <name xml:lang="zh">黄河花店</name>
  <name xml:lang="en">Huang He Hwa Dian</name>
    .....
</businessEntity>
```

The following shows an example of transliteration where the primary name of the business is in Chinese, and is a transliteration of its alternative English name:

```
<businessEntity . . . >
  ........
  <name xml:lang="zh">康柏電腦股份有限公司</name>
  <name xml:lang="en">Compaq Computer Taiwan Limited</name>
    .....
</businessEntity>
```

The following example XML fragment shows an address written in two languages, English and Chinese, as indicated by the xml:lang attribute:

```
<address useType="Sales office" xml:lang="en" tModelKey="uddi:…">
   <addressLine>7 F</addressLine>
   <addressLine>No. 245 </addressLine>
   <addressLine>Sec. 1</addressLine>
   <addressLine>Tunhua South Road</addressLine>
   <addressLine>Taipei </addressLine>
</address>
</address><address useType="Sales office" xml:lang="zh" tModelKey="uddi:…">
   <addressLine> 台北市 </addressLine>
   <addressLine> 敦化南路</addressLine>
   <addressLine> 一段</addressLine>
   <addressLine> 245 號</addressLine>
   <addressLine> 7 樓</addressLine>
   …
</address>
```

D.2 Multiple names in the same language

In order to support acronyms or multi-script languages, it is valid to publish multiple names that have identical language identification.

The following shows an example of use of multiple name elements to support a multi-script language and also the use of acronym. In the example, the first <name> element is the primary name of the business (a Japanese flower shop) in Japanese Kanji. The second <name> element is the business' name transliterated into Japanese Katakana. The third <name> element gives the business' full English name, and the fourth <name> element gives its English acronym:

```
<businessEntity . . . >
  ........
  <name xml:lang="ja">日本生花店</name>
  <name xml:lang="ja">ニッポンセイカテン</name>
  <name xml:lang="en">NIPPON FLOWERS </name>
  <name xml:lang="en">NF</name>
  .....
</businessEntity>
```

Where multiple name elements are published, the first name element is treated as the primary name, which is the name by which a business would be searched and sorted in the case of multiply-named businesses. Client applications may use this knowledge to assist in optional rendering of a publisher's primary name or all alternative names.

D.3 Internationalized address format

The <address> element, contained in the businessEntity structure, contains a simple list of <addressLine> elements.

To expose an address' structure and meaning, virtual keyedReference elements are employed. This is done by adorning the <address> element with a tModelKey attribute and use of the keyName/keyValue attribute pair for each <addressLine> element.

Additionally, the UDDI Business Registry has a canonical tModel, ubr-uddi-org:postalAddress, that identifies a canonical postal address structure with common address sub-elements (e.g. states, cities). This canonical address structure describes address data via name/code pairs, enabling each common address sub-element to be identified by name or code.

The following example XML fragment shows how the application of tModelKey, keyName and keyValue attributes to <address>, in conjunction with the use of address sub-element names

and codes defined by the ubr-uddi-org:postalAddress tModel, allows the structure and meaning of a contact's address within a businessEntity to be derivable programmatically:

```
<address useType="Sales office" tModelKey="uddi:uddi.org:ubr:postaladdress">
   <addressLine keyName="Street" keyValue="60">Alexanderplatz</addressLine>
   <addressLine keyName="House number" keyValue="70">12</addressLine>
   …
   <addressLine keyName="Country" keyValue="20">Deutschland</addressLine>
</address>
```

The following example XML fragment shows an address in two languages where the sequence of the address lines differ according to the language used. With the use of keyName/KeyValue pair together with the codes assigned in the ubr-uddi-org:postalAddress tModel, it is possible to determine the address semantics programmatically in spite of the difference in address sequence :

```
<address useType="Sales office" xml:lang="en"
tModelKey="uddi:uddi.org:ubr:postaladdress">
   <addressLine keyName="Floor" keyValue="100">7 F</addressLine>
   <addressLine keyName="House Number" keyValue="70">No. 245 </addressLine>
   <addressLine keyName="District" keyValue="50">Sec. 1</addressLine>
   <addressLine keyName="Street" keyValue="60">Tunhua South Road</addressLine>
   <addressLine keyName="City" keyValue="40">Taipei </addressLine>
</address>
</address><address useType="Sales office" xml:lang="zh"
tModelKey="uddi:uddi.org:ubr:postaladdress">
   <addressLine keyName="City" keyValue="40"> 台北市 </addressLine>
   <addressLine keyName="Street" keyValue="60"> 敦化南路</addressLine>
   <addressLine keyName="District" keyValue="50"> 一段</addressLine>
   <addressLine keyName="House Number" keyValue="70"> 245 號</addressLine>
   <addressLine keyName="Floor" keyValue="100"> 7 樓</addressLine>
   …
</address>
```

As there is a large variation in address sub-elements of different countries[53], the defined canonical address structure does not attempt to include all possible address sub-elements of all countries. Freeform address lines are therefore supported in the <address> element.

The usage of the canonical address structure is optional, but recommended, for both publishers of business entities and developers of GUIs of UDDI publishing services.

[53] For information on other address format standardization efforts, refer to standards bodies, such as the Universal Postal Union and ECCMA, which defines an International Address Element Code (IAEC).

D.4 Language–dependent collation

The UDDI specifications allow the collation sequence of results returned by the Inquiry APIs to be specified via find qualifiers. The following is an example tModel overviewDoc that illustrates the specification of a language-specific sort order tModel based on a language-specific collation standard, in this case, the JIS X 4061 Japanese Collation Sequence.

D.4.1 UDDI JIS X 4061 Japanese Sort Order Qualifier

D.4.1.1 Introduction

The sortOrder type of find qualifier (a subset of find qualifier) represents a collation sequence applied to the result set. The JIS X 4061 Japanese sortOrder find qualifier directs that a sort be performed on the result set elements according to the JIS X 4061 - 1996 "Collation of Japanese Character Strings" standard.

D.4.1.2 Design Goals

The JIS X 4061 Japanese sortOrder tModel is provided to enable inquiry results to be sorted according to the JIS X 4061 - 1996 "Collation of Japanese Character Strings" standard.

D.4.1.3 tModel Definition

This tModel is a find qualifier that is used to enable sorting of UDDI inquiry results, based on the JIS X 4061 - 1996 "Collation of Japanese Character Strings" standard, for Hiragana and Katakana characters. All other characters will be sorted according to the Default Unicode Collation Element Table. When this tModel is referenced in a find qualifier, a sort is performed on the field designated by the sortBy* find qualifier (name, by default), normalized using Unicode Normalization Form C. This qualifier conflicts with any other find qualifier of the sortOrder type, such as uddi.org:binarySort.

Name: ubr-uddi-org:JIS-X4061

Short name: JIS-X4061

Description: UDDI JIS X 4061 Japanese collation sequence find qualifier

UDDI Key (V3): uddi:ubr-uddi.org:sortorder:jis-x4061

Categorization: sortOrder, findQualifier

Support: Optional

D.4.1.4 tModel Structure

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:uddi.org:sortorder:jis-x4061">
  <name>uddi-org:JIS-X4061</name>
  <description>UDDI JIS X 4061 Japanese
               collation sequence find qualifier
  </description>
  <overviewDoc typeURI="text">
    <overviewURL>
        http://ubr.uddi.org/overviewDocs/UBR_CoreOther_tModels.doc#JIS-X4061Sort
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:categorization"
      keyValue="sortOrder"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:categorization"
      keyValue="findQualifier"
      tModelKey="uddi:uddi.org:categorization:types"/>
   </categoryBag>
</tModel>
```

D.4.1.5 Example of Use

The following represents a typical inquiry that references the JIS X 4061 Japanese sortOrder find qualifier tModel. This example finds businesses or their contained businessServices or bindingTemplates that are categorized with any value using the 'tempuri-org:CustomerType' value set. The businessEntities so found are sorted first by name in ascending order, using the JIS X 4061 Japanese collation sequence, and for those that share a common name, by descending order of the date of the most recently updated entity.

```
<find_business xmlns="urn:uddi-org:api_v3">
  <findQualifiers>
   <findQualifier>
       uddi:uddi.org:sortorder:jis-4061
   </findQualifier>
   <findQualifier>
       uddi:uddi.org:findqualifier:sortbydatedesc
   </findQualifier>
   <findQualifier>
       uddi:uddi.org:findqualifier:sortbynameasc
   </findQualifier>
   <findQualifier>
       uddi:uddi.org:findqualifier:approximatematch
   </findQualifier>
   <findQualifier>
       uddi:uddi.org:findqualifier:combinecategorybags
   </findQualifier>
  </findQualifiers>
  <categoryBag>
    <keyedReference keyValue="%"
      keyName="tempuri-org:CustomerType"
      tModelKey="uddi:uddi.org:categorization:general_keywords"/>
  </categoryBag>
</find_business>
```

E Appendix E: Using Identifiers

One of the design goals associated with the UDDI registration data is the ability to mark information with identifiers. The purpose of identifiers in the UDDI registration data, namely for businessEntity and tModel instances, is to allow others to find the published information using more formal identifier systems. For example, businesses may want to use their D-U-N-S® number[54], Global Location Number (GLN)[55], or tax identifier in their UDDI registration data, since these identifiers are shared in a public or private community in order to unambiguously identify businesses. In UDDI registries that are only used in private communities, businesses may also want to use privately known identifiers. For example, in a UDDI registry that is used as a service registry in a private exchange, supplier identifiers that are only known in this community might be used to identify the businesses.

When looking at an identifier, such as a D-U-N-S® number, it is not always immediately apparent what the identifier represents. For instance, consider the following identifier:

12-345-6789

Standing alone, it is not possible to figure out what this combination of digits and formatting characters implies. However, if it is known that this is a D-U-N-S® number, it is at least clear that this string identifies a business. Therefore, all appearances of identifiers in UDDI registries pair the identifier itself with its identifier system as in the following example.

D-U-N-S® Number, 12-345-6789

E.1 Using identifiers

Two of the main UDDI data structure types provide a structure to support attaching identifiers to data. These are the businessEntity and the tModel structures. By providing a placeholder for attaching identifiers to these data structures, any number of identifiers can be used for a variety of purposes.

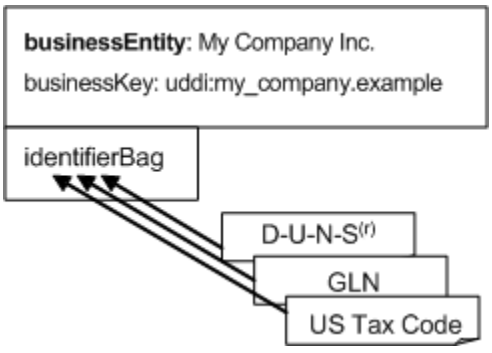


Figure 5 - Using Identifiers

[54] D-U-N-S® Numbers are provided by Dun & Bradstreet. See http://www.dnb.com.

[55] The Global Location Number system is defined in the EAN UCC system (http://www.ean-int.org/locations.html).

In the example shown in Figure 5, the businessEntity with the name "My Company Inc." specified three identifiers in its identifierBag. These identifiers can be used, for example, in a find_business call in order to locate the businessEntity in the UDDI registry.

For instance, it is likely that someone who wants to find the types of technical Web services that are exposed by a given business would search using a business identifier. In the example shown in Figure 5 the individual who registered the businessEntity data specified a D-U-N-S® number, a Global Location Number, and a US Tax Code identifier[56].

The following fragment of an XML document shows an example of how an identifier is added to a businessEntity in its identifierBag.

```
<businessEntity businessKey="uddi:my_company.example">
    …
    <identifierBag>
        <keyedReference
            tModelKey="uddi:uddi.org:ubr:identifier:dnb.com:d-u-n-s"
            keyName="D-U-N-S:My Company"
            keyValue="12-345-6789"/>
        …
    </identifierBag>
    …
</businessEntity>
```

The businessEntity instance that is technically identified with the businessKey uddi:my_company.example contains an identifierBag with a D-U-N-S number[57]. This is established by the three attributes of the keyedReference:

- **tModelKey**: uniquely identifies the tModel that represents the identifier system
- **keyName**: human readable name of the identifier system, and when the identity is coded, a human readable rendition of the value
- **keyValue**: the actual identifier within the specified identifier system

If a registry follows the recommended policy for recognizing identifier systems, all identifier systems that are registered with a specific UDDI registry can be discovered with the find_tModel call as follows:

```
<find_tModel>
    <categoryBag>
        <keyedReference
            tModelKey="uddi:uddi.org:categorization:types"
            keyValue="identifier"/>
    </categoryBag>
</find_tModel>
```

[56] In the diagram, the actual name/value properties were abbreviated for the sake of simplicity.

[57] The identifier system examples in this section refer to actual tModelKeys that will be used to identify the corresponding tModels in the UDDI Business Registry, Version 3.

F Appendix F: Using Categorization

Besides the ability to mark UDDI registration data with identifiers, another design goal is the ability to assign category information. Without categorization, locating data within a UDDI registry would prove to be very difficult.

Especially for the discovery of previously unknown businesses, services, bindings or service types, it is indispensable that the corresponding UDDI registration data is marked with a set of categories that can universally be searched on. For example, the Universal Standard Products and Services Classification (UNSPSC)[58], a set of categorization codes representing product and service categories, can be used to specify a business' product and service offering in a more formalized way.

F.1 Using simple categories

All four main UDDI data structure types provide a structure to support attaching categories to data. These are the businessEntity, businessService, bindingTemplate and the tModel structures. By providing a placeholder for attaching categories to these data structures, any number of categories can be used for a variety of purposes.

The following fragment of an XML document shows an example[59] of how categories are added to a businessEntity using a categoryBag.

```
<businessEntity businessKey="uddi:my_company.example">
    …
    <categoryBag>
        <keyedReference
            tModelKey="uddi:uddi.org:ubr:categorization:unspsc"
            keyName="UNSPSC:Medical Equipment and Accessories and Supplies"
            keyValue="42.00.00.00.00"/>
        <keyedReference
            tModelKey="uddi:uddi.org:ubr:categorization:unspsc"
            keyName="UNSPSC:Drugs and Pharmaceutical Products"
            keyValue="51.00.00.00.00"/>
        <keyedReference
            tModelKey="uddi:uddi.org:ubr:categorization:iso3166"
            keyName="GEO:Germany"
            keyValue="DE"/>
        <keyedReference
            tModelKey="uddi:uddi.org:ubr:categorization:iso3166"
            keyName="GEO:France"
            keyValue="FR"/>
        <keyedReference
            tModelKey="uddi:uddi.org:ubr:categorization:iso3166"
            keyName="GEO:United States"
            keyValue="US"/>
    </categoryBag>
</businessEntity>
```

The businessEntity instance that is identified with the businessKey uddi:my_company.example contains a categoryBag with two UNSPSC product category codes and three ISO 3166[60]

[58] See http://eccma.org/unspsc.

[59] The category system examples in this section refer to actual tModelKeys that are used to identify the corresponding tModels in the UDDI Business Registry, Version 3.

country codes. The business that is represented by this businessKey wants to specify that it sells medical equipment and pharmaceutical products in Germany, France and the United States. This is technically established by the three attributes of each of the keyedReferences:

- **tModelKey**: uniquely identifies the tModel that represents the category system
- **keyName**: human readable name of the category system, and when the actual category is coded, a human readable rendition of the value
- **keyValue**: the actual category code within the specified category system

In order to find all category systems that are registered with a specific UDDI registry that follows the recommended policy for recognizing category systems, the find_tModel call can be used as follows:

```
<find_tModel>
    <categoryBag>
        <keyedReference
            tModelKey="uddi:uddi.org:categorization:types"
            keyValue="categorization"/>
    </categoryBag>
</find_tModel>
```

In order to simply use keywords instead of full-fledged category systems, the UDDI general keywords taxonomy, as specified in Chapter 11, *Utility tModels and Conventions*, can be used.

```
<businessEntity businessKey="uddi:my_company.example">
    …
    <categoryBag>
        <keyedReference
            tModelKey="uddi:uddi.org:categorization:general_keywords"
            keyName="example.org:ConsultingTypes:Web service consulting"
            keyValue="Web services"/>
        <keyedReference
            tModelKey="uddi:uddi.org:categorization:general_keywords"
            keyName="example.org:ConsultingTypes:UDDI consulting"
            keyValue="UDDI"/>
    </categoryBag>
</businessEntity>
```

The businessEntity instance that is identified with the businessKey uddi:my_company.example contains a categoryBag with two keywords by using the UDDI general keywords categorization system. The business that is represented by this businessKey wants to specify that it offers Web service and UDDI consulting services by using the keywords "Web service" and "UDDI" as keyValue attributes within the keyedReference elements.

[60] See http://www.iso.org/iso/en/prods-services/iso3166ma/index.html.

F.2 Grouping categories

For many cases, the use of single categories is adequate for describing the characteristics of a business, a service, a binding, or a tModel so that it can easily be found.

But there are some cases where the relationship between single categories becomes important. Taking the example from the section above, the business might want to specify that it actually sells medical equipment only in Germany and France and pharmaceutical products only in the United States.

For this and similar purposes, categoryBags can contain keyedReferenceGroups that in turn contain a list of keyedReferences. Since the set of keyedReferences that are grouped within a keyedReferenceGroup do not themselves provide any meaning why they are grouped together, the keyedReferenceGroup carries its own tModelKey identifying a tModel that in turn provides this meaning.

The following XML fragment shows how keyedReferenceGroups are used in order to achieve the desired behavior.

```
<businessEntity businessKey="uddi:my_company.example">
    …
    <categoryBag>
        <keyedReferenceGroup
            tModelKey=
                "uddi:uddi.org:ubr:categorizationgroup:unspsc_geo3166">
            <keyedReference
                tModelKey="uddi:uddi.org:ubr:categorization:unspsc"
                keyName="UNSPSC:Medical Equipment and Accessories and Supplies"
                keyValue="42.00.00.00.00"/>
            <keyedReference
                tModelKey="uddi:uddi.org:ubr:categorization:iso3166"
                keyName="GEO:Germany"
                keyValue="DE"/>
        </keyedReferenceGroup>
        <keyedReferenceGroup
            tModelKey=
                "uddi:uddi.org:ubr:categorizationgroup:unspsc_geo3166">
            <keyedReference
                tModelKey="uddi:uddi.org:ubr:categorization:unspsc"
                keyName="UNSPSC:Medical Equipment and Accessories and Supplies"
                keyValue="42.00.00.00.00"/>
            <keyedReference
                tModelKey="uddi:uddi.org:ubr:categorization:iso3166"
                keyName="GEO:France"
                keyValue="FR"/>
        </keyedReferenceGroup>
        <keyedReferenceGroup
            tModelKey=
                "uddi:uddi.org:ubr:categorization:unspsc_geo3166">
            <keyedReference
                tModelKey="uddi:uddi.org:ubr:categorization:unspsc"
                keyName="UNSPSC:Drugs and Pharmaceutical Products"
                keyValue="51.00.00.00.00"/>
            <keyedReference
                tModelKey="uddi:uddi.org:ubr:categorization:iso3166"
                keyName="GEO:United States"
                keyValue="US"/>
        </keyedReferenceGroup>
        …
    </categoryBag>
</businessEntity>
```

The tModel that is used in this example, keyed as "uddi:uddi.org:ubr:categorizationgroup:unspsc_geo3166", represents the grouping of the

category systems for UNSPSC and ISO 3166. As a consequence, keyedReferenceGroups that reference this tModel describe countries or regions where a product category is offered.

Another use case that shows the importance of grouping categories is the idea of using geographical coordinates in order to specify where a specific business or service is physically located. The following example details how geographical latitudes and longitudes are grouped together in a keyedReferenceGroup. The geographic reference system used in this example is the World Geodetic System 1984 (WGS 84).

```
<businessEntity businessKey="uddi:my_company.example">
    <categoryBag>
        <keyedReferenceGroup
            tModelKey="uddi:uddi.org:ubr:categorizationGroup:wgs84">
            <keyedReference
                tModelKey=
                    "uddi:uddi.org:ubr:categorization:wgs84:latitude"
                keyName="WGS 84 Latitude"
                keyValue="+49.682700"/>
            <keyedReference
                tModelKey=
                    "uddi:uddi.org:ubr:categorization:wgs84:longitude"
                keyName="WGS 84 Longitude"
                keyValue="+008.295200"/>
            <keyedReference
                tModelKey=
                    "uddi:uddi.org:ubr:categorization:geo_precision"
                keyName="Center of Street"
                keyValue="0900"/>
        </keyedReferenceGroup>
    </categoryBag>
</businessEntity>
```

The businessEntity instance that is identified with the businessKey uddi:my_company.example contains a categoryBag with one keyedReferenceGroup that in turn contains a latitude, a longitude, and a precision information grouped using the WGS 84 system. The business that is represented by this businessKey wants to specify that it is physically located at the latitude/longitude pair 49.6827/8.2952 and that this positioning information was derived for the center of the street the business is being located in.

In order to find all category group systems that are registered within a UDDI registry that follows the recommended policy for recognizing category group systems, the find_tModel call can be used as follows:

```
<find_tModel>
    <categoryBag>
        <keyedReference
            tModelKey="uddi:uddi.org:categorization:types"
            keyValue="categorizationGroup"/>
    </categoryBag>
</find_tModel>
```

In order to use category group systems that are not yet registered with a specific UDDI registry, the publisher of the category group system can register this as a new tModel.

F.3 Deriving categories

In order to use value sets, such as geographic taxonomies, for different purposes, value sets can be derived from the tModels for some other value sets to reuse the sets of values. Each of these derived value sets represents a specific meaning that is described in its name and overviewDoc. The following example shows how a business is categorized with the same sets of values, but for different purposes. The first keyedReference shows the categorization for the physical location of the business. The second shows the categorization for the service area, the default meaning of the geo3166-2 value set.

```
<businessEntity businessKey="uddi:my_company.example">
    …
    <categoryBag>
        <keyedReference
            tModelKey=
                "uddi:uddi.org:ubr:categorization:iso3166:location"
            keyName="GEOLocation:California"
            keyValue="US-CA"/>
        <keyedReference
            tModelKey="uddi:uddi.org:ubr:categorization:iso3166"
            keyName="GEO:United States"
            keyValue="US"/>
    </categoryBag>
</businessEntity>
```

In order to find all category systems that are derived from a base category system, taking the ISO 3166 category system as an example, and registered with a UDDI registry that follows the recommended policy for identifying derived value sets, the find_tModel call can be used as follows:

```
<find_tModel>
    <categoryBag>
        <keyedReference
            tModelKey="uddi:uddi.org:categorization:derivedfrom"
            keyValue="uddi:uddi.org:ubr:categorization:iso3166"/>
    </categoryBag>
</find_tModel>
```

Note that the result list for an inquiry using the derivedFrom category system consists of all category systems that are derived from the value set specified in the keyValue. In order to find all derived category systems, regardless of their base category system, in a registry that follows to the recommended policy for identifying derived value sets, the following find_tModel call can be used:

```
<find_tModel>
    <findQualifiers>
      <findQualifier>
         uddi:uddi.org:findqualifier:approximatematch
      </findQualifier>
    </findQualifiers>
    <categoryBag>
        <keyedReference
            tModelKey="uddi:uddi.org:categorization:derivedfrom"
            keyValue="%"/>
    </categoryBag>
</find_tModel>
```

The root category systems are themselves categorized with the valueSet uddi type. To find all of the root value sets in a registry that follows the recommended policy for identifying root value sets, the following find_tModel call can be used:

```
<find_tModel>
    <categoryBag>
        <keyedReference
            tModelKey="uddi:uddi.org:categorization:types"
            keyValue="valueSet"/>
    </categoryBag>
</find_tModel>
```

G Appendix G: Wildcards

This appendix provides examples of how to use wildcards in various search operations using the find_xx APIs, as described in Section 5.1.6 *About Wildcards*.

G.1 Find using "starts with" searching

To find all businesses whose name begins with "ABC" – e.g., "ABC Vacuum", the following find_business can be used:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<find_business xmlns = "urn:uddi-org:api_v3"
xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance">
  <findQualifiers>
    <findQualifier>
      uddi:uddi.org:findqualifier:approximatematch
    </findQualifier>
  </findQualifiers>
  <name>ABC%</name>
</find_business>
```

G.2 Find using "starts and ends with" searching

To find all the businesses whose name begins with "Texas" and ends with "Cafe" the following find_business can be used:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<find_business xmlns = "urn:uddi-org:api_v3"
xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance">
  <findQualifiers>
    <findQualifier>
      uddi:uddi.org:findqualifier:approximatematch
    </findQualifier>
  </findQualifiers>
  <name>Texas%Cafe</name>
</find_business>
```

G.3 Find using escaped literals

To find all the businesses whose name contains a literal "_" the following find_business can be used:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<find_business xmlns = "urn:uddi-org:api_v3"
xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance">
  <findQualifiers>
    <findQualifier>
      uddi:uddi.org:findqualifier:approximatematch
    </findQualifier>
  </findQualifiers>
  <name>%\_%</name>
</find_business>
```

G.4 Find using wildcards with Taxonomies

To find all businesses classified using the D-U-N-S number system the following find_business can be used:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<find_business xmlns = "urn:uddi-org:api_v3"
xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance">
  <findQualifiers>
    <findQualifier>
      uddi:uddi.org:findqualifier:approximatematch
    </findQualifier>
  </findQualifiers>
  <identifierBag>
    <keyedReference
      keyValue = "%"
      tModelKey = "uddi:uddi.org:ubr:identifier:dnb.com:d-u-n-s"/>
  </identifierBag>
</find_business>
```

To find all businesses classified in any of the UNSPSC categories in the UNSPSC family "Telephones and personal telecommunications devices and accessories" the following find_business can be used:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<find_business xmlns = "urn:uddi-org:api_v3"
xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance">
  <findQualifiers>
    <findQualifier>
      uddi:uddi.org:findqualifier:approximatematch
    </findQualifier>
  </findQualifiers>
  <categoryBag>
    <keyedReference
       keyValue = "34.10.__.__.00"
       tModelKey = "uddi:uddi.org:ubr:categorization:unspsc"/>
  </categoryBag>
</find_business>
```

H Appendix H: Extensibility

UDDI supports extension through a derivation mechanism provided by XML Schema to enable access to additional functionality using extended UDDI API and data structures. Using XML schema derivation to extend UDDI retains compatibility with the core UDDI specification. XML Schema restrictions are explicitly prohibited to prevent the UDDI core functionalities from being arbitrarily limited.

Other than allowing extensibility in the XML schema, the UDDI schema extension mechanism does not affect any other parts of the specification. There are, however, some implied behaviors worth pointing out:

- If a client sends a request with extension to a UDDI registry that does not support the extension, the registry MUST return an error. The registry MAY indicate the namespace of the extension that is not supported in the error text.
- If a UDDI data structure is extended using XML schema extensions in a multi-node registry, the extended elements MUST be replicated by all nodes. This is implied by the definition of a registry that requires all nodes to host the same data, subject to replication latency. See Section 1.5 *Base UDDI Architecture* for details.

Before introducing an extension, the designer of an extension SHOULD consider the impact of the extension. The designer SHOULD document the impact so that administrators and users of the registry can be fully aware of the impact in deciding whether to adopt an extension or not.

H.1 Using the basic UDDI infrastructure

UDDI provides many levels and dimensions of flexibility such as arbitrary categorization schemes, open-ended discoveryURLs, etc. Many requirements can be satisfied by some application of the specification without introducing any extension. UDDI technical notes and best practices are good sources of information about such applications of the UDDI specification.

H.2 Establishing an extension

H.2.1 Extension designer

Extensions to UDDI can be accomplished by extending UDDI data structures (see Chapter 3 *UDDI Data Structures*) and/or extending the UDDI API (See Chapter 5 *UDDI Programmers APIs*). Each schema extension is given its own namespace. Furthermore, the XML schema xsi:type substitutionGroup SHOULD be used to represent the extension elements.

The designer of a UDDI extension SHOULD also register one or more tModel(s) to represent the extension. Each tModel SHOULD indicate that it is an extension of one or more UDDI native API set(s) by containing a categorization entry indicating the uddi:uddi.org:categorization:derivedfrom categorization tModel, with the keyValue being the tModelKey of a UDDI native API tModel. The categorization allows clients to discover extensions of a UDDI native API.

For example, an extension that extends the UDDI v3 inquiry API is represented by the following keyedReference entry in a categoryBag

```
<keyedReference
  keyName="uddi-org:derivedFrom:v3_inquiry"
  keyValue="uddi:uddi.org:v3_inquiry"
  tModelKey="uddi:uddi.org:categorization:derivedfrom"/>
```

H.2.2 Registries that support the extension

Conceptually speaking, a UDDI registry is a Web service. A Web service that supports the UDDI v3 specification and a Web service that supports UDDI v3 specification with the extension are two distinct Web services. Hence, a UDDI registry that implements the extension SHOULD provide two sets of service end points: one set that supports the UDDI v3 specification and one set that supports the UDDI v3 specification with the extension.

To allow clients to establish the functional differences between the two sets of service end points, the registry SHOULD indicate the differences in the service end point bindingTemplates. The additional service end point(s) that support the extension SHOULD reference tModels for the extension.

H.2.2.1 Special considerations in data structure extension

In the case of data structure extension, if the registry chooses to support one set of service end point(s) instead of two sets, there will be some undesirable consequences. Specifically, if the registry receives a get_xx request, the registry cannot distinguish whether the client is prepared to handle the extension or not, because in a data structure extension, the get_xx API is not extended. With two sets of service end points, the registry can identify whether the client is prepared to handle the extension based upon the end point which the client uses.

As an alternative, an extension designer can also choose to extend the correspond save_xx and get_xx API set so that one set of service end point(s) is sufficient to cover both the native API and the extension.

H.3 Programmers API and UDDI Clients

There are two types of UDDI Clients: UDDI clients that are not prepared to handle the extension and UDDI clients that are prepared to handle the extension.

H.3.1 UDDI Clients not prepared to handle the extension

A UDDI registry that supports an extension must handle native UDDI clients that have no knowledge of or interest in the extension. Clients of this type will refer to the UDDI namespaces contained in Chapter 2 *UDDI Schemas* in their API calls. The registry must respond to such client requests by following the specification for the native UDDI namespace, performing the behavior prescribed in this specification.

If a client follows a save of an extended entity with a subsequent save of the entity without its extension, extension documentation determines whether to remove the extension information or leave it untouched.

H.3.2 UDDI Clients prepared to handle the extension

A UDDI registry that receives a message referring to a namespace for an extension that it supports should respond with the appropriate extended response structure, if one exists. If the API itself is extended, the registry should process the message according to the extended behavior.

H.4 Error Codes

A UDDI extension may introduce additional error conditions. Whenever possible, extensions SHOULD reuse existing error codes to minimize the impact on client interoperability. See Chapter 12 *Error Codes* for more information.

H.5 Digital signatures

Digital signatures are used to provide integrity and authenticity of the data managed in a UDDI registry. All of the UDDI core entities support adding XML digital signatures.

If an entity extension can be signed, either by carrying its own signature element or by being covered by the signature of the extended entity, the designer of the extension SHOULD provide guidance. Signing an extension provides a guarantee of integrity on the data, but it may introduce difficulties with interoperability between registries in entity promotion scenarios.

It is worth pointing out that a publisher can also choose to provide two signatures, one for the entity without the extension and one for the entity with the extension. In this case, however, a publisher then needs additional transforms to exclude all the signature elements from the entity being signed. A digital signature standard transform can only exclude the enveloped signature itself, but not its peers.

If the extension affects save_xx API calls, the extension SHOULD NOT alter the entity that a client sends because altering an entity can create unnecessary difficulties for a client who digitally signs the entity.

In the case where the extension MAY alter the entity in a save_xx API call, the extension designer SHOULD document this behavior.

H.6 Entity promotion

If the extension extends UDDI data structures, there can be some complications when an entity is promoted from one UDDI registry to another UDDI registry, if the two registries do not support the same set of extensions. The owners of the two registries should agree on the extensions that will be promoted through some out-of-band communication. Additional tools may be required to transform or filter out some extensions.

Any transformation or filtering of the entity may invalidate the digital signatures it contains. Hence, any UDDI data structure extensions should be introduced with great caution.

H.7 Replication

In a multi-node UDDI registry where all nodes support the same data structure extension, there are no replication issues. Taking advantage of a property of XML schema derivation, regular change records can contain the extended data structure under the extension namespace. The other nodes in the registry, upon receiving such change records, will be able to recognize the data structure extension and save it correctly.

Using a data structure extension may limit the ability to add a new UDDI node from a different vendor, however, since the extension may not be supported by that vendor.

H.8 Example

The following example illustrates the above considerations and provides a template for documenting an extension. We will assume that the document is located at http://tempuri.org/uddi_extension/specification.html. This URL is needed in the extension tModel.

H.8.1 Description

A discoveryURLs element is added to the UDDI publisherAssertion structure. This extension enables a business relationship to reference additional supporting documentation, such as a business agreement between the two parties. Additional documentation of a business relationship is particularly useful in a community / marketplace usage scenario.

H.8.2 Data structure (XML schema)

A new schema is created to extend the UDDI Version 3 API schema. In this example, the normative schema document is located at http://tempuri.org/schema/uddi_v3PublisherAssertionExt.xsd.

Note the targetNamespace and the egExt namespace definition. This schema imports the UDDI Version 3 API schema, which allows extension through schema derivation.

A new type named *publisherAssertionExt* is defined to extend the *publisherAssertion* element in the UDDI Version 3 API schema. The extension is indicated by the *xsd:extension* element. An optional *discoveryURLs* element is added to the *publisherAssertion* element. When a registry that supports this schema extension receives a publisherAssertion structure that references this extended namespace, that does not include the optional *discoveryURLs* element, it MUST behave just as it does without the extension.

```
<xsd:schema
    targetNamespace="http://tempuri.org/uddi_extension"
    xmlns:egExt="http://tempuri.org/uddi_extension"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:uddi="urn:uddi-org:api_v3"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">
  <xsd:import namespace="urn:uddi-org:api_v3"
    schemaLocation="http://uddi.org/schema/uddi_v3.xsd" />
  <xsd:element name="publisherAssertionExt"
              type="egExt:publisherAssertionExt"
              substitutionGroup="uddi:publisherAssertion"/>
  <xsd:complexType name="publisherAssertionExt">
     <xsd:annotation>
        <xsd:documentation>A complex type which is an extension of the
publisherAssertion complex type in UDDI.
        </xsd:documentation>
     </xsd:annotation>
     <xsd:complexContent>
        <xsd:extension base="uddi:publisherAssertion">
             <xsd:sequence>
                <xsd:element ref="uddi:discoveryURLs"
                   minOccurs = "0"/>
             </xsd:sequence>
        </xsd:extension>
     </xsd:complexContent>
  </xsd:complexType>
</xsd:schema>
```

H.8.3 tModel of the extension

The extension designer then publishes two tModels to identify Web services that comply with the UDDI extension, one for the extended inquiry API and one for the extended publication API. The uddi:uddi.org:categorization:derivedfrom category is used to represent the derivation.

tModel name: tempuri-org:v3_inquiry:publisherAssertionExt

tModel Description: An extended UDDIv3 inquiry API, with an extension of UDDI publisherAssertions to allow discoveryURLs in a publisherAssertion.

tModel key: uddi:tempuri.org:v3_inquiry:publisherassertionext

Categorization: specification, xmlSpec, soapSpec

Derived from (tModelKeys): uddi:uddi.org:v3_inquiry

This tModel is an extension to the UDDI Version 3 inquiry API (defined by tModel uddi-org:inquiry_v3), enabling discoveryURLs to be included in a publisherAssertion. For more information, see http://tempuri.org/uddi_extension/specification.html.

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:tempuri.org:v3_inquiry:publisherassertionext">
  <name>tempuri-org:v3_inquiry:publisherAssertionExt</name>
  <description>
    Extension of UDDI publisherAssertion to allow
    discoveryURLs in a keyedReference.
  </description>
  <overviewDoc>
    <overviewURL useType="text">
           http://tempuri.org/uddi_extension/specification.html
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:soapSpec"
      keyValue="soapSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:xmlSpec"
      keyValue="xmlSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:specification"
      keyValue="specification"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference
      keyName="uddi-org:derivedFrom:v3_inquiry"
      keyValue="uddi:uddi.org:v3_inquiry"
      tModelKey="uddi:uddi.org:categorization:derivedfrom"/>
   </categoryBag>
</tModel>
```

tModel name: tempuri-org:v3_publication:publisherAssertionExt

tModel Description: An extended UDDIv3 publication API, with an extension of UDDI publisherAssertions to allow discoveryURLs in a publisherAssertion.

tModel key: uddi:tempuri.org:v3_publication:publisherassertionext

Categorization: specification, xmlSpec, soapSpec

Derived from (tModelKeys): uddi:uddi.org:v3_publication

This tModel is an extension to the UDDI Version 3 publication API (defined by uddi-org:publication_v3), enabling discoveryURLs to be included in a publisherAssertion. For more information, see http://tempuri.org/uddi_extension/specification.html.

This tModel is represented with the following structure:

```
<tModel tModelKey="uddi:tempuri.org:v3_publication:publisherassertionext">
  <name>tempuri-org:v3_publication:publisherAssertionExt</name>
  <description>
    Extension of UDDI publisherAssertion to allow
    discoveryURLs in a keyedReference.
  </description>
  <overviewDoc>
    <overviewURL useType="text">
         http://tempuri.org/uddi_extension/specification.html
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference keyName="uddi-org:types:soapSpec"
      keyValue="soapSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:xmlSpec"
      keyValue="xmlSpec"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference keyName="uddi-org:types:specification"
      keyValue="specification"
      tModelKey="uddi:uddi.org:categorization:types"/>
    <keyedReference
      keyName="uddi-org:derivedFrom:v3_publication"
      keyValue="uddi:uddi.org:v3_publication"
      tModelKey="uddi:uddi.org:categorization:derivedfrom"/>
   </categoryBag>
</tModel>
```

H.8.4 Additional service end points

A registry SHOULD provide a pair of inquiry and publication service end points in bindingTemplate elements that support the base UDDI v3 specification, along with an additional pair of inquiry and publication service end points in bindingTemplate elements which support the extension.

H.8.5 Programmers API Description of the extension

A publisherAssertion can contain discoveryURLs.

For add_publisherAssertions, and set_publisherAssertions, the argument publisherAssertion is modified as follows:

- ***publisherAssertion***: one or more relationship assertions. Relationship assertions consist of a reference to two businessEntity key values as designated by the fromKey and toKey elements, as well as a required statement of the directional relationship within the contained keyedReference element. See Appendix A *Relationships and Publisher Assertions* for more information. The fromKey, the toKey, and all three parts of the keyedReference – the tModelKey, the keyName, and the keyValue – must be specified. Empty (zero length) keyNames and keyValues are permitted. Furthermore, when using publisherAssertionExt elements of type *publisherAssertionExt* in the *egExt* namespace, discoveryURLs can be optionally provided.

For get_publisherAssertions and get_assertionStatusReport, if a request is made to the publication endpoint that supports the extension, the response message can contain a publisherAssertion of the type egExt:publisherAssertionExt. If a request is made to the publication endpoint that does not support extension, the response message must contain publisherAssertion of the native UDDI v3 type.

For find_relatedBusinesses, there is no change in the request message. The syntax and the matching rules for the keyedReference remain the same. The response message can contain a publisherAssertion of the type egExt:publisherAssertionExt when the namespace used on the API is egExt.

For normal get_publisherAssertions, get_assertionStatusReport, and find_relatedBusinesses APIs, a request sent to a native UDDI API service end point MUST return publisherAssertion elements without the extension. A request sent to an extended service end point MUST return extended publisherAssertion elements, if available.

H.8.6 Digital signature

publisherAssertion elements may be signed. Depending on the usage scenario, a publisher may include or exclude the extension element in signing. A publisher may also provide two signatures, one that includes the extension and one that excludes the extension.

If a publisher chooses to exclude the extension element in signing, the following XSLT transform can be used:

```
<dsig:Transform xmlns:dsig="http://www.w3.org/2000/09/xmldsig#"
     Algorithm="http://www.w3.org/TR/1999/REC-xslt-19991116">
   <xsl:stylesheet
         xmlns:xsl="http://www.w3.org/TR/1999/REC-xslt-19991116"
         xmlns:egExt="http://tempuri.org/uddi_extension"
         xmlns:uddi="urn:uddi-org:api_v3"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <xsl:output mode="xml" />
      <!-- use publisherAssertion in uddi v3 namespace and
           exclude the extension element uddi:discoveryURLs -->
      <xsl:template match="egExt:publisherAssertionExt"
                    priority="1">
          <uddi:publisherAssertion>
             <xsl:apply-templates select="uddi:fromKey" />
             <xsl:apply-templates select="uddi:toKey" />
             <xsl:apply-templates select="uddi:keyedReference" />
          </uddi:publisherAssertion>
      </xsl:template>

      <!-- default identity transformation -->
      <xsl:template match="@*|node()" priority="0">
          <xsl:copy>
              <xsl:apply-templates select="@*|node()"/>
          </xsl:copy>
      </xsl:template>

   </xsl:stylesheet>
</dsig:Transform>
```

H.8.7 Registry operation: replication

All nodes must support the extension to the extent that the all nodes must replicate the extended publisherAssertion elements of the type egExt:publisherAssertionExt.

H.8.8 Registry operation: entity promotion

The policy decision makers for the two registries should agree on whether the extension will be promoted through some out-of-band communication.

If the target registry supports the extension and the extension is promoted, then there is no issue.

If the target registry does not support the extension, the extension MAY by filtered out by removing the discoveryURLs element and redefining the publisherAssertionExt element to be of the UDDI native publisherAssertion type.

Such filtering, however, may impact digital signatures. There are two scenarios:

- If a publisher signs a publisherAssertion with a transform that excludes the extension, there is no issue. The manifest used to generate the digital signature does not consist of the extension. Hence, the digital signature of the publisherAssertion in the target registry remains valid.
- If a publisher signs a publisherAssertion without any additional transforms, the digital signature of the publisherAssertion in the target registry will be invalid since the digital signature covers the extension.

H.8.9 Non-normative example

This example demonstrates the addition of a publisherAssertion with a pointer to the contract between the two entities contained in the extended publisherAssertion element. The publisherAssertionExt element is defined to be of the type egExt:publisherAssertionExt (as opposed to the UDDI native publisherAssertion type uddi:publisherAssertion).

```
<uddi:add_publisherAssertions
        xmlns:egExt="http://tempuri.org/uddi_extension"
        xmlns:uddi="urn:uddi-org:api_v3"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <uddi:authInfo>someAuthInfo</uddi:authInfo>
   <egExt:publisherAssertionExt>
      <uddi:fromKey>some business key</uddi:fromKey>
      <uddi:toKey>some other business key</uddi:toKey>
      <uddi:keyedReference
            tModelKey="uddi:uddi.org:relationships"
            keyName="some peer to peer relationship"
            keyValue="peer-peer" />
      <uddi:discoveryURLs>
         <uddi:discoveryURL useType="contract">
            http://www.example.com/contract/p2pcontract.pdf
         </uddi:discoveryURL>
      </uddi:discoveryURLs>
   </egExt:publisherAssertionExt>
</uddi:add_publisherAssertions>
```

Now we perform an inquiry to obtain all of the publisherAssertions owned by the publisher, including the extension. In order to obtain the extension, a client must send the request to the publication service end point that supports the extension.

```
<get_publisherAssertions
    xmlns="urn:uddi-org:api_v3">
  <authInfo>the-authinfo-token</authInfo>
</get_publisherAssertions>
```

Next we perform an inquiry for all of the publisherAssertions owned by the publisher, without the extension. A client must send the request to the publication service end point that supports only the native UDDI publication API set.

```
<get_publisherAssertions
    xmlns="urn:uddi-org:api_v3">
  <authInfo>the-authinfo-token</authInfo>
</get_publisherAssertions
```

In entity promotion, if the target registry filters out the extension, the extended publisherAssertion will be transformed to the UDDI native publisherAssertion. The discoveryURLs element is removed.

```
<uddi:add_publisherAssertions
        xmlns:uddi="urn:uddi-org:api_v3">
   <uddi:authInfo>someAuthInfo</uddi:authInfo>
   <!-- egExt:publisherAssertionExt element
        is replaced by uddi:publisherAssertion element -->
   <uddi:publisherAssertion>
      <uddi:fromKey>some business key</uddi:fromKey>
      <uddi:toKey>some other business key</uddi:toKey>
      <uddi:keyedReference
            tModelKey="uddi:uddi.org:relationships"
            keyName="some peer to peer relationship"
            keyValue="peer-peer" />
      <!-- uddi:discoveryURLs element is removed -->
   </uddi:publisherAssertion>
</uddi:add_publisherAssertions>
```

I Appendix I: Support For XML Digital Signatures

The UDDI v3 schema supports signing of the following UDDI elements using XML-Signature Syntax and Processing (see http://www.w3.org/TR/xmldsig-core/).

- businessEntity
- businessService
- bindingTemplate
- tModel
- publisherAssertion

This Appendix describes the process for signing the UDDI elements listed above using XML Digital Signature and the process for verifying signatures associated with these elements.

I.1 Generation of a Signature

A Signature element SHOULD be generated according to the required steps of "Core Generation" in XML-Signature Syntax and Processing.

The signature should be calculated on the top level element that will be stored by the registry as a result of the Publication API call. This element, referred to as the data object in the XML-Signature and Syntax specification, is the businessEntity element for save_business API calls, the businessService element for save_service API calls, the bindingTemplate for save_binding API calls, the tModel for save_tModel API calls and the publisherAssertion for set_publisherAssertions and add_publisherAssertions API calls. The signature should be generated on the elements before they are added to the body of an API call. Also, according to the signature generation, all children of the element being signed are included in the generation of the signature unless first excluded by application of a transform. For businessService elements, any bindingTemplate elements must be included in the signature generation unless first excluded by applying a transform. Similarly, for businessEntity elements, businessService elements must be included in the signature generation unless first excluded by applying a transform. Due to the containment of service projections as businessService elements within a businessEntity element, this also means that changes to the projected service will render a signature of the businessEntity containing the projection invalid, unless a businessService element representing a service projection is excluded using a transform.

Because attributes are also included in generation of a signature on an element, unless excluded by applying a transform, node generated entity keys result in the addition of content in attribute values. Publication API calls that include data where the keys are not assigned by the publisher will not be able to generate a signature that will remain valid, unless the node assigned key attributes are excluded using a transform. It is RECOMMENDED that publishers generate a signature on elements containing publisher assigned keys. In the event that publisher assigned keys cannot be used due to the registry or node policy on key generation, it is RECOMMENDED that publishers generate a signature on elements after all keys in the element and its contained elements have been generated by the node. For example, when signing a businessService element, both the serviceKey and businessKey MUST be provided, and in any contained bindingTemplate elements, both the bindingKey attributes and the serviceKey attributes must be provided on each bindingTemplate. The generation of a signature where node generated keys are included in the signature is, then, only possible on updates of the data where no new keys are to be generated by the node.

Due to the location of the sequence of Signature elements within an element that is to be signed, the signature is "enveloped". As a result of the enveloping of the signature, it is necessary to apply at least one transformation on the signed entity to exclude the signature or signature(s). The transformation selected by a publisher or the XML Signature tool is specified in a Transform element inside the Signature element. In the case that a publisher is generating only one signature per containing element, the "Enveloped Signature Transform" specified in XML-Signature Syntax and Processing is an appropriate transform to support the enveloping.

After the "Reference Generation" steps are performed according to XML-Signature Syntax and Processing, the SignedInfo should be generated, requiring the selection of a Canonicalization Method. It is strongly RECOMMENDED that the Schema Centric Canonicalization algorithm be used for producing the canonical form of the data object referenced in the SignedInfo. Other canonicalization algorithms, including the Canonical XML referenced in the XML-Signature Syntax and Processing specification, do not account for the nature of the response structures in UDDI registries. Due to the re-enveloping, namespace prefix normalization, Unicode normalization and other schema based alterations to the data objects, signatures generated using other canonicalization algorithms MAY NOT and likely will not validate successfully when the entity is retrieved from one or more nodes of a registry.

It is RECOMMENDED that well known key formats be used in the KeyInfo applied during generation of the signature. As indicated by XML Signature Syntax and Processing, Section 4.4, "questions of trust of such key information (e.g., its authenticity or strength) are out of scope of this specification and left to the application." It is then left to the publisher to produce a signature where the credentials asserted in the KeyInfo MAY be verified by an inquirer. Without diligent verification of the contents and validity of any published KeyInfo in the Signature element, XML-Signature Syntax and Processing only provides integrity with respect to the data supplied by the publisher but provides no assurance of the identity of the publisher.

I.2 Validation of a Signature

A Signature element returned in the get_xxDetails API call SHOULD be validated according to the required steps of "Core Validation" in XML-Signature Syntax and Processing.

The input to the signature validation should be the top level element that is returned by the registry as a result of the Inquiry API call. This element, referred to as the data object in the XML-Signature and Syntax specification, is the businessEntity element for get_businessDetail API calls, the businessService element for get_serviceDetail API calls, the bindingTemplate for get_bindingDetail API calls, the tModel for get_tModelDetail API calls and the publisherAssertion contained in the sharedRelationship structure in find_relatedBusinesses API calls.

Typically, the nodes will not validate the signature on behalf of a client, so any client that uses signed data for any form of assurance MUST validate the signatures on the relevant data objects and MUST perform the necessary diligence on the KeyInfo if the inquirer uses the KeyInfo element for an assurance of identity of the publisher.

J Appendix J: UDDI Replication Examples

J.1 Communication Graph

The following diagram is an example of a simple communicationGraph that restricts the invocation of get_changeRecords messages to a unidirectional-ring amongst a set of four nodes. In operational practice, it is expected that this get_changeRecords message will typically be the only UDDI replication message where communication restrictions are imposed. The introduction of communication restrictions on the notify_changeRecordsAvailable messages between nodes within a registry is not expected to be commonplace.

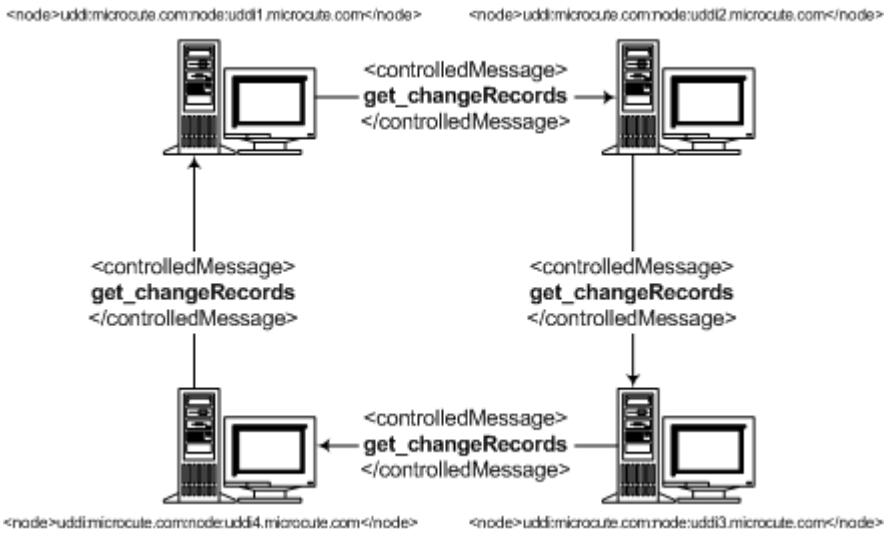


Figure 6 - Communication Graph Example

The arrows in Figure 6 depict the direction of flow of datum from one node to another.

The communication graph depicted in the Replication Configuration Structure example in the next section represents a spanning cycle topology. This is one of many possible topologies supported within this replication framework.

J.2 Replication Configuration Structure Example

```
<?xml version="1.0" encoding="UTF-8"?>
<replicationConfiguration xmlns="urn:uddi-org:repl_v3">
   <serialNumber>8</serialNumber>
   <timeOfConfigurationUpdate>200203041859Z</timeOfConfigurationUpdate>
   <registryContact>
        <contact xmlns="urn:uddi-org:api_v3">
            <description>Registry Administrative Contact</description>
            <personName>King Kurt</personName>
            <phone useType="Voice">425-555-1212</phone>
            <email useType="e-mail">kkurt@microcute.com</email>
        </contact>
   </registryContact>
```

```
    <operator>
    <operatorNodeID>uddi:microcute.com:node:uddi1.microcute.com</operatorNodeID>
         <operatorStatus>normal</operatorStatus>
         <contact useType="Operator Contact" xmlns="urn:uddi-org:api_v3">
             <description>Microcute Customer Contact</description>
             <personName>Customer Support</personName>
             <phone useType="Voice">444.555.5589</phone>
             <email useType="e-mail">uddiOpsExample@microcute.com</email>
         </contact>
         <operatorCustodyName>www.microcute.com/svcs/uddi</operatorCustodyName>
         <soapReplicationURL>
             https://www.microcute.com/svcs/uddi/repl.asmx
         </soapReplicationURL>
    </operator>
    <operator>

    <operatorNodeID>uddi:microcute.com:node:uddi2.microcute.com</operatorNodeID>
         <operatorStatus>normal</operatorStatus>
…
    </operator>
    <operator>

    <operatorNodeID>uddi:microcute.com:node:uddi3.microcute.com</operatorNodeID>
         <operatorStatus>normal</operatorStatus>
…
    </operator>
    <operator>

    <operatorNodeID>uddi:microcute.com:node:uddi4.microcute.com</operatorNodeID>
         <operatorStatus>normal</operatorStatus>
…
    </operator>
<communicationGraph>
         <node>uddi:microcute.com:node:uddi1.microcute.com</node>
         <node>uddi:microcute.com:node:uddi2.microcute.com</node>
         <node>uddi:microcute.com:node:uddi3.microcute.com</node>
         <node>uddi:microcute.com:node:uddi4.microcute.com</node>
         <controlledMessage>get_changeRecords</controlledMessage>
         <edge>
             <message>get_changeRecords</message>
             <messageSender>
             uddi:microcute.com:node:uddi1.microcute.com
         </messageSender>
             <messageReceiver>
             uddi:microcute.com:node:uddi2.microcute.com
         </messageReceiver>
         </edge>
         <edge>
             <message>get_changeRecords</message>
             <messageSender>
             uddi:microcute.com:node:uddi2.microcute.com
         </messageSender>
             <messageReceiver>
             uddi:microcute.com:node:uddi3.microcute.com
         </messageReceiver>
         </edge>
         <edge>
             <message>get_changeRecords</message>
             <messageSender>
             uddi:microcute.com:node:uddi3.microcute.com
         </messageSender>
             <messageReceiver>
             uddi:microcute.com:node:uddi4.microcute.com
         </messageReceiver>
         </edge>
         <edge>
             <message>get_changeRecords</message>
             <messageReceiver>
             uddi:microcute.com:node:uddi4.microcute.com
         </messageReceiver>
             <messageReceiver>
```

```
            uddi:microcute.com:node:uddi1.microcute.com
        </messageReceiver>
        </edge>
</communicationGraph>
   <maximumTimeToGetChanges>12</maximumTimeToGetChanges>
</replicationConfiguration>
```

J.3 notify_changeRecordsAvailable Example

The following is an example of a notify_changeRecordsAvailable call.

```
<?xml version="1.0" encoding="UTF-8"?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
   <Body>
<notify_changeRecordsAvailable xmlns="urn:uddi-org:repl_v3">
   <notifyingNode>
        uddi:microcute.com:node:uddi2.microcute.com
   </notifyingNode>
   <changesAvailable>
        <highWaterMark>
            <nodeID>
                uddi:microcute.com:node:uddi2.microcute.com
            </nodeID>
            <originatingUSN>123</originatingUSN>
        </highWaterMark>
        <highWaterMark>
            <nodeID>
                uddi:microcute.com:node:uddi1.microcute.com
            </nodeID>
            <originatingUSN>241</originatingUSN>
        </highWaterMark>
        <highWaterMark>
            <nodeID>
                uddi:microcute.com:node:uddi3.microcute.com
            </nodeID>
            <originatingUSN>193</originatingUSN>
        </highWaterMark>
        <highWaterMark>
            <nodeID>
                uddi:microcute.com:node:uddi4.microcute.com
            </nodeID>
            <originatingUSN>173</originatingUSN>
        </highWaterMark>
   </changesAvailable>
</notify_changeRecordsAvailable>
   </Body>
</Envelope>
```

J.4 get_ChangeRecords Example

The following is an example of a get_changeRecords call.

```
<?xml version="1.0" encoding="UTF-8"?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
   <Body>
        <get_changeRecords
                xmlns:xsd=http://www.w3.org/2001/XMLSchema
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xmlns="urn:uddi-org:repl_v3">
           <requestingNode>
                uddi:microcute.com:node:uddi1.microcute.com
           </requestingNode>
           <changesAlreadySeen>
                <highWaterMark>
                    <nodeID>
                        uddi:microcute.com:node:uddi1.microcute.com
                    </nodeID>
                    <originatingUSN>242</originatingUSN>
                </highWaterMark>
                <highWaterMark>
                    <nodeID>
                        uddi:microcute.com:node:uddi2.microcute.com
                    </nodeID>
                    <originatingUSN>120</originatingUSN>
                </highWaterMark>
                <highWaterMark>
                    <nodeID>
                        uddi:microcute.com:node:uddi3.microcute.com
                    </nodeID>
                    <originatingUSN>193</originatingUSN>
                </highWaterMark>
                <highWaterMark>
                    <nodeID>
                        uddi:microcute.com:node:uddi4.microcute.com
                    </nodeID>
                    <originatingUSN>172</originatingUSN>
                </highWaterMark>
           </changesAlreadySeen>
        </get_changeRecords>
   </Body>
</Envelope>
```

J.5 Miscellaneous Replication Example

The following XML Instance document describes several replication specific messages.

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <soap:Body>
        <changeRecords xmlns="urn:uddi-org:repl_v3">
            <changeRecord acknowledgementRequested="false">
                <changeID>
                    <nodeID>uddi:microcute.com:node:uddi2.microcute.com</nodeID>
                    <originatingUSN>120</originatingUSN>
                </changeID>
                <changeRecordNewData>
                    <businessEntity businessKey="
                            uddi:microcute.com:uddi2:microcute.com:3f10…"
                            operator="Microcute UDDI Services"
                            authorizedName="…"
                            xmlns="urn:uddi-org:api_v3">
                        <discoveryURLs>
                            <discoveryURL useType="homepage">
                                http://...</discoveryURL>
                            <discoveryURL useType="businessEntity">
                                http://....</discoveryURL>
                        </discoveryURLs>
                        <name xml:lang="en">Simple Business</name>
                        <description xml:lang="en">Simple Business</description>
                        <businessServices>
                            <businessService serviceKey="…"
                                    businessKey="uddi:microcute.com:3f10…">
                                <name xml:lang="en">Off Shore Mining</name>
                                <description xml:lang="en">
                                    Off shore drilling
                                </description>
                                <bindingTemplates>
                                    <bindingTemplate bindingKey="…."
                                        serviceKey="…">
                                        <accessPoint URLType="https">
                                            http://...</accessPoint>
                                    </bindingTemplate>
                                </bindingTemplates>
                                <categoryBag>
                                    <keyedReference tModelKey="…"
                                        keyName="Iron Ores"
                                        keyValue="1010"/>
                                </categoryBag>
                            </businessService>
                        </businessServices>
                    </businessEntity>
                </changeRecordNewData>
            </changeRecord>
            <changeRecord acknowledgementRequested="false">
                <changeID>
                    <nodeID>uddi:microcute.com:node:uddi2.microcute.com</nodeID>
                    <originatingUSN>121</originatingUSN>
                </changeID>
                <changeRecordNewData>
                    <tModel tModelKey="…"
                            operator="Microcute UDDI Services"
                            authorizedName="…"
                            xmlns="urn:uddi-org:api_v3">
                        <name>Simple tModel</name>
                        <overviewDoc>
```

```
                            <overviewURL>http://…</overviewURL>
                        </overviewDoc>
                        <identifierBag>
                            <keyedReference tModelKey="…"
                                keyName="Lab Tested Protocol"
                                keyValue="protocol"/>
                        </identifierBag>
                        <categoryBag>
                            <keyedReference tModelKey="…"
                                keyName="Protocol"
                                keyValue="protocol"/>
                        </categoryBag>
                    </tModel>
                </changeRecordNewData>
            </changeRecord>
            <changeRecord acknowledgementRequested="true">
                <changeID>
                    <nodeID>uddi:microcute.com:node:uddi2.microcute.com</nodeID>
                    <originatingUSN>122</originatingUSN>
                </changeID>
                <changeRecordHide>
                    <tModelKey xmlns="urn:uddi-org:api_v3">….</tModelKey>
                </changeRecordHide>
            </changeRecord>
            <changeRecord acknowledgementRequested="false">
                <changeID>
                    <nodeID>uddi:microcute.com:node:uddi2.microcute.com</nodeID>
                    <originatingUSN>123</originatingUSN>
                </changeID>
                <changeRecordPublisherAssertion>
                    <publisherAssertion>
                        <fromKey>….</fromKey>
                        <toKey>….</toKey>
                        <keyedReference tModelKey="…"
                            keyName="Holding Company"
                            keyValue="parent-child"/>
                    </publisherAssertion>
                    <fromBusinessCheck>true</fromBusinessCheck>
                    <toBusinessCheck>false</toBusinessCheck>
                </changeRecordPublisherAssertion>
            </changeRecord>
        </changeRecords>
    </soap:Body>
</soap:Envelope>
```

J.6 Non-normative – Cycle of Cycles Topology

This section describes one of many possible topology choices allowed by this Replication Design. This appendix is an illustrative example only.

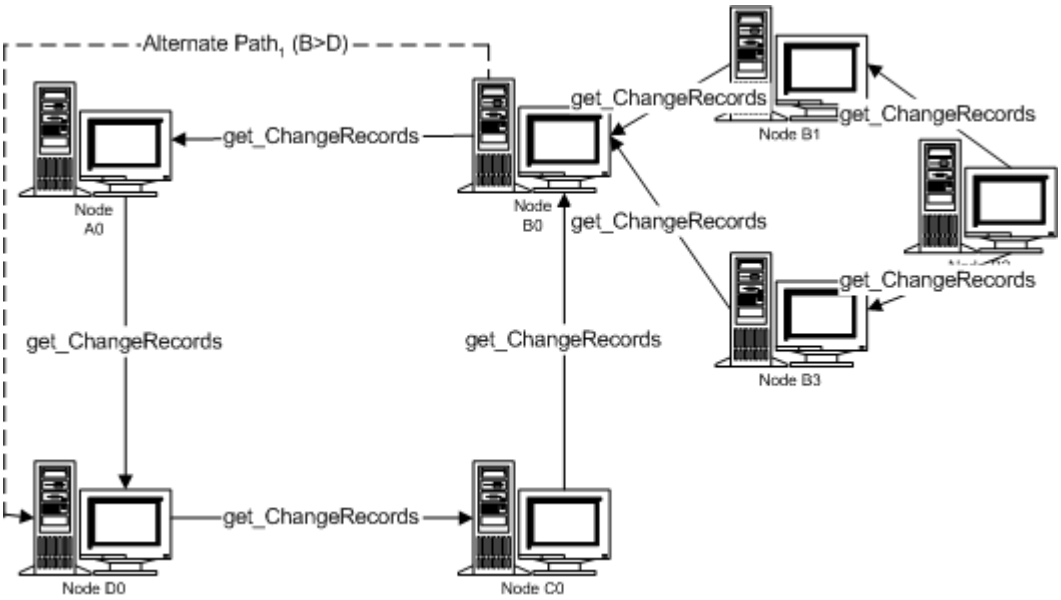


Figure 7 - Cycle of Cycles Topology

If the Communications Graph were a cycle of cycles, (e.g., if in addition to the ABCD spine, A0 was also on a cycle with A1, A2, and A3, B0 was on a cycle with B1, B2, and B3, etc.) a variation of this would be applied. When a spine node such as B0 receives a get_changeRecords request (or in the case of A, when its timer event is triggered), it handles propagation through its sub-cycle before answering the get_changeRecords request on the "spine". B0 continues propagation along the spine only after the first round of propagation completes on its sub-cycle, which means that B has received all changes from B1, B2, and B3. It does not continue the second round of propagation on its sub-cycle until it receives the second round of changes from A, which contains updates from all other spine nodes and their sub-cycles.

For a cycle, it is suggested that a backup node will be the next node in the cycle, and for added safety, the node after that as well, allowing propagation to proceed correctly even if there are node failures. Note that an incorrectly perceived node failure will not affect the correctness of the algorithm. If additional propagation is done, no unnecessary data will be replicated, although there may be additional get_changeRecords within the prescribed period

K Appendix K – Modeling UDDI within UDDI – A Sample

UDDI itself is a set of Web services. With this in mind, modeling UDDI within UDDI is an important and illuminating exercise to help understand modeling decisions that need to be made when using UDDI. Moreover, the modeling of UDDI within UDDI is a required practice for all operators of UDDI nodes. For a complete explanation of recommended vs. required modeling decisions for UDDI nodes, see Chapter 9 *Policy*.

This appendix will walk through the recommended modeling of a multi-versioned instance of UDDI. Each fragment of the different modeling pieces will be outlined, with an explanation of the salient modeling decisions and requirements made below.

K.1 The Node's businessEntity

All nodes must create a Node Business Entity, under which the various services they offer are modeled.

K.1.1 XML Fragment

```
<businessEntity
        businessKey="uddi:tempuri.org:uddinodebusinessKey"
        xmlns="urn:uddi-org:api_v3">
   <name xml:lang="en">A UDDI Node</name>
   <description xml:lang="en">This represents a sample model of how a uddi node
        might represent itself in UDDI
   </description>
   <categoryBag>
        <keyedReference tModelKey="uddi:uddi.org:categorization:nodes"
            keyValue="node" />
   </categoryBag>
```

K.1.2 Explanation

As explained in Section 6.2.2.1 *Normative Modeling of Node Business Entity,* a node must categorize its Node Business Entity with the **uddi:uddi.org:categorization:nodes** category, using the keyValue of **node**.

Also note the creation of a UDDI v3 businessKey with the domainKey of uddi:tempuri.org:uddinodebusinessentity.

K.2 The Policy Service

With v3, there are a number of policy decision points that a node must document. These policies should be documented using the Policy schema. Policies placed in the document should reflect overarching policies for the node. Each specific API set may have an additional policy statement. Or, the entirety of a node's policy might be in the XML file denoted by the Policy service.

K.2.1 XML Fragment

```
<businessService
        serviceKey="uddi:tempuri.org:policyservicekey"
        businessKey="uddi:tempuri.org:nodebusinesskey">
   <name xml:lang="en">UDDI Policy Service</name>
   <description xml:lang="en">Web Service supporting UDDI policy
        information
   </description>
   <bindingTemplates>
        <bindingTemplate bindingKey="uddi:tempuri.org:uddinodepolicy"
                serviceKey="uddi:tempuri.org:policyservicekey">
            <description xml:lang="en">This binding provides an HTTP GET
                XML document outlining overall policy decision points for
                this node.
            </description>
            <accessPoint useType="endPoint">
                https://tempuri.org/uddi/overall_policy.xml
            </accessPoint>
            <tModelInstanceDetails>
                <tModelInstanceInfo tModelKey="uddi:uddi.org:v3_policy">
                    <description xml:lang="en">This binding supports the
                        UDDI Version 3.0 policy modeling schema.
                    </description>
                </tModelInstanceInfo>
                <tModelInstanceInfo
                    tModelKey="uddi:uddi.org:protocol:serverauthenticatedssl3">
                    <description xml:lang="en">This binding's access point
                        requires the use of SSL 3.0 with server
                        authentication.
                    </description>
                </tModelInstanceInfo>
            </tModelInstanceDetails>
        </bindingTemplate>
   </bindingTemplates>
</businessService>
```

K.2.2 Explanation

First, notice that there is a Policy businessService with a single bindingTemplate underneath it as a child. Also, note the keys for the policy businessService and bindingTemplate are based on the same domainKey.

The accessPoint for the policy file is an XML file called overall_policy.xml. The useType attribute on the accessPoint has a value of "text", which denotes that there is no indirection required to retrieve this data.

There are two tModelInstanceInfos which the Policy bindingTemplate implement. First is the UDDI v3 Policy tModel. Implementing this tModel signifies that the XML file returned by this accessPoint conforms to the specification laid out in the uddi:uddi.org:v3_Policy tModel.

The second tModel denotes that there is server authenticated SSL required to access this resource. One could determine this by parsing the address itself for the **https** prefix, but by modeling as such, this information can be determined through queries using the UDDI API.

K.2.2.1 Sample Policy Document

The following document is an example of a policy document. Not all policy decision points have been demonstrated in this sample, but rather, only a subset is represented. As shown in the example, the policy document for the policy service includes both registry-defined policies and policies delegated to the node.

```
<?xml version="1.0" encoding="UTF-8" ?>
<policies xmlns="urn:uddi-org:policy_v3">
   <policy>
   <policyName>Registering Nodes in a Registry</policyName>
        <policyDescription xml:lang="en">
            This registry contains two nodes, the tempuri.org node and the
            example.org node. The configuration
            of these nodes is protected by a signed Replication Configuration
            Structure.
            Access to this configuration file is only available to the operators of
            the nodes. Nodes may be added by mutual agreement of both nodes.
            In the event that either node is removed from the registry,
            publishers will receive notification and all data custody will be
            transferred to one the remaining nodes chosen by the publisher.
        </policyDescription>
   <policyDecisionPoint>registry</policyDecisionPoint>
   </policy>
   <policy>
        <policyName>User Limits</policyName>
        <policyDescription xml:lang="en">
            There are two levels of publisher accounts for the tempuri.org node.
            Level 1:
            A level 1 publisher account is used by individual businesses and
            organizations registering at the http://tempuri.org/uddi site.

            Current limits are as follows:

            1 Business Entity;
            4 Business Services per Business Entity;
            2 Binding Templates per Business Service;
            10 tModels;

            In the event that a publisher needs to register additional information,
            an account upgrade with increased limits for business entities or
            tModels may be requested by sending an email to node@tempuri.org.
            Additional information about the publisher will be collected to help
            verify requirements to register additional information.

            Level 2:

            A level 2 publisher account is typically used by large organizations,
            marketplaces, or service providers that provide registration services
            on behalf of multiple businesses. These accounts have no restrictions
            on the amount of information that may be registered within the node.

            A publisher account may be upgraded from Level 1 to Level 2 by
            contacting the node. To request a publisher account upgrade, send an
            email request to node@tempuri.org.

        </policyDescription>
        <policyDecisionPoint>node</policyDecisionPoint>
   </policy>
   <policy>
        <policyName>Audit</policyName>
        <policyDescription xml:lang="en">
            The http://tempuri.org/uddi site will monitor and audit all
            publication activity at its site. Any publisher registered at the node
            may request the history of its published information by sending an
            email request to node@tempuri.org. The history of each registered entry
```

```
                published at the node will be provided within one week.

                Audit history will be maintained for one year.
            </policyDescription>
            <policyDecisionPoint>node</policyDecisionPoint>
        </policy>
        <policy>
            <policyName>Data Integrity</policyName>
            <policyDescription xml:lang="en">
                This registry will endeavor to maintain the integrity of all
                information registered at any node of the registry, either through
                the programmatic XML and SOAP interfaces, or any other interface
                provided by a node. Limitations imposed by the specification include,
                but are not limited to, the following:

                    Whitespace will be normalized and trimmed according
                    to the rules defined in the UDDI specification and
                    schemas.  All content will also be normalized according
                    to the Unicode normalization algorithm referenced in
                    the specification.

                    Size limits for individual data elements stored within
                    the UDDI registry have been established in the
                    specification. Information registered at any node
                    exceeding these size limits will be rejected.

                The terms of use statement for publication of information at
                http://tempuri.org/uddi requires that all information published be
                accurate, and that the publisher is authorized to represent their
                individual organization(s). Should violations of this policy be
                detected, and/or unauthorized publication of information occur,
                notification of such violation should be sent via email to
                node@tempuri.org. The information in the registry will be tracked
                internally as suspect or contested. The registered contact information
                for the publisher of the contested information will be provided within
                4 business hours.

                Final resolution of the conflict is responsibility of the parties
                involved, and should follow established legal processes. The
                http://tempuri.org/uddi operations team will comply with the resolution
                agreed to by the parties involved, or the results of any subsequent
                litigation related to the issue. This compliance is limited to
                removal of the contested information from the UDDI registry.
            </policyDescription>
            <policyDecisionPoint>node</policyDecisionPoint>
        </policy>
    </policies>
```

K.3 The Security Service

In UDDI Version 3, the Security API is an optional API that may or may not be supported by a node. In this sample, the node does support the security API, and has indicated this by modeling a specific security service, where clients may use the Security API.

K.3.1 XML Fragment

```
<businessService
   serviceKey="uddi:tempuri.org:authservicekey"
   businessKey="uddi:tempuri.org:nodebusinesskey">
   <name xml:lang="en">UDDI Authentication Service</name>
   <description xml:lang="en">
        Web Service supporting UDDI Security API
   </description>
   <bindingTemplates>
        <bindingTemplate
                bindingKey="uddi:tempuri.org:authbinding"
                serviceKey="uddi:tempuri.org:authservicekey">
            <description xml:lang="en">
                This binding to authenticate with the UDDI services using the UDDI
                Security API.
            </description>
            <accessPoint useType="endPoint">
                https://tempuri.org/uddi/authenticate.asmx
            </accessPoint>
            <tModelInstanceDetails>
                <tModelInstanceInfo tModelKey="uddi:uddi.org:v3_security">
                    <description xml:lang="en">This binding's supports the UDDI v3
                        Security API.
                    </description>
                </tModelInstanceInfo>
                <tModelInstanceInfo
                    tModelKey="uddi:uddi.org:protocol:serverauthenticatedssl3">
                    <description xml:lang="en">
                        This binding's access point requires the use of SSL 3.0
                        with server authentication.
                    </description>
                </tModelInstanceInfo>
            </tModelInstanceDetails>
        </bindingTemplate>
   </bindingTemplates>
</businessService>
```

K.3.2 Explanation

Similar to the policy businessService, the security businessService has a single bindingTemplate which contains the accessPoint for the Security API itself.

The accessPoint provides the end point where this API can be invoked. It too is decorated with a useType attribute of "endPoint", denoting that there is no indirection involved in the accessPoint.

The bindingTemplate implements the uddi:uddi.org:v3_security API, and thus complies with the requirements which are represented by that tModel.

Similar to the Policy tModel, this Security API models its security protocol using the **uddi:uddi.org:protocol:serverauthenticatedssl3** tModel.

K.4 The Publish Service – Supporting 3 Versions

In this sample, the node supports all three UDDI Publish API sets. This has been modeled as a single Publish businessService with three different bindingTemplates, one for each version of the UDDI Publish API it supports.

K.4.1 XML Fragment

```
<businessService
        serviceKey="uddi:tempuri.org:publishservicekey"
        businessKey="uddi:tempuri.org:nodebusinesskey">
   <name xml:lang="en">UDDI Publish API Services</name>
   <description xml:lang="en">Web Service supporting UDDI
specifications</description>
   <bindingTemplates>


<bindingTemplate
        bindingKey="uddi:tempuri.org:uddinodebindingkey_publish_v1"
        serviceKey="uddi:tempuri.org:publishservicekey">
   <description xml:lang="en">
        This binding supports the UDDI Programmer's API Specification for
        publication
   </description>
   <accessPoint useType="endPoint">
        https://tempuri.org/uddi/publish.asmx
   </accessPoint>
   <tModelInstanceDetails>
        <tModelInstanceInfo
                tModelKey="uddi:64c756d1-3374-4e00-ae83-ee12e38fae63">
            <description xml:lang="en">
                This binding supports the UDDI Version 1.0 Programmer's API
                Specification for publication
            </description>
        </tModelInstanceInfo>
   </tModelInstanceDetails>
</bindingTemplate>


<bindingTemplate
        bindingKey="uddi:tempuri.org:uddinodebindingkey_publish_v2"
        serviceKey="uddi:tempuri.org:publishservicekey">
   <description xml:lang="en">
        This binding supports the UDDI Programmer's API Specification for
        publication
   </description>
   <accessPoint useType="endPoint">
        https://tempuri.org/uddi/publish_v2.asmx
   </accessPoint>
   <tModelInstanceDetails>
        <tModelInstanceInfo
                tModelKey="uddi:a2f36b65-2d66-4088-abc7-914d0e05eb9e">
            <description xml:lang="en">
                This binding supports the UDDI Version 2.0 Programmer's API
                Specification for publication
            </description>
        </tModelInstanceInfo>
   </tModelInstanceDetails>
</bindingTemplate>


<bindingTemplate
        bindingKey="uddi:tempuri.org:uddinodebindingkey_publish_v3"
        serviceKey="uddi:tempuri.org:publishservicekey">
   <description xml:lang="en">
        This binding supports the UDDI Programmer's API Specification for
        publication
   </description>
```

```
    <accessPoint useType="endPoint">
         https://tempuri.org/uddi/publish_v3.asmx
    </accessPoint>
    <tModelInstanceDetails>
         <tModelInstanceInfo tModelKey="uddi:uddi.org:v3_publication">
             <description xml:lang="en">
                 This binding supports the UDDI Version 3.0 Programmer's API
                 Specification for publication
             </description>
             <instanceDetails>
                 <overviewDoc>
                     <description xml:lang="en">
                         This overviewURL provides policy information about
                         publication.
                     </description>
                     <overviewURL useType="text">
                         https://tempuri.org/uddi/publish_policy.xml
                     </overviewURL>
                 </overviewDoc>
                 <instanceParms>
                     <![CDATA[
                         <?xml version="1.0" encoding="utf-8" ?>
                             <UDDIinstanceParmsContainer
                                     xmlns="urn:uddi-org:policy_v3_instanceParms">
                                 <authInfoUse>required</authInfoUse>
                             </UDDIinstanceParmsContainer>
                     ]]>
                 </instanceParms>
             </instanceDetails>
         </tModelInstanceInfo>
         <tModelInstanceInfo
                 tModelKey="uddi:uddi.org:protocol:serverauthenticatedssl3">
             <description xml:lang="en">
                 This binding's access point requires the use of SSL 3.0 with server
                 authentication.
             </description>
         </tModelInstanceInfo>
    </tModelInstanceDetails>
</bindingTemplate>


    </bindingTemplates>
</businessService>
```

K.4.2 Explanation

First, note the creation of three bindingTemplates with domainKeys for each of the different API versions: **uddi:tempuri.org:uddinodebindingkey_publish_v1, uddi:tempuri.org:uddinodebindingkey_publish_v2, and uddi:tempuri.org:uddinodebindingkey_publish_v3.** In this sample, there must be three different bindingTemplates because each accessPoint for the different APIs is different. Were a node to support multiple versions through the same accessPoint, fewer bindingTemplates would be required for the purposes of modeling.

The Version 1 and Version 2 bindingTemplates are identical except for two differences: the accessPoints differ and the tModelInstanceInfo differs, as each bindingTemplate implements a different version of the API. Note that the Version 1 and Version 2 bindingTemplates implement their publish API version tModels with a UUID. Because these canonical tModels do not have a corresponding v3 domainKey, the UUIDs are taken from the earlier version of the specification and used.

In the Version 3 bindingTemplate, note the instanceDetails of the tModelInstanceInfo for the v3 API set. There is an overviewDoc specified that directs a user to policy information about that API specifically. For example, this document might discuss publication limits, processes for gaining a publication account, etc. It is not required that a separate policy document be

created for each API, but it is convenient for the user to be able to directly access policy information about that API.

Also, there is an XML fragment using <![CDATA[…]]> within the instanceParms. This XML fragment models the authInfo use policy for this API, as explained in Chapter 9 *Policy*. In this case, an authInfo element is required on all Publish API calls to this node.

The Version 3 bindingTemplate also implements the SSL tModel, as it must be invoked through SSL.

Note that the Version 1 and 2 APIs are not modeled to say that they implement the SSL tModel, nor do they have a policy overviewURL. This modeling decision was made because the use of SSL is specified by the definition of the UDDI Version 1 and 2 Publish API sets and because the conventions for modeling the use of SSL did not exist until the Version 3 specification. However, a node may choose to model these newer canonical tModels on older API sets as appropriate.

K.5 The Inquiry Service – Supporting 3 Versions

Lastly, the node has modeled three different versions of the Inquiry API under a single logical businessService.

K.5.1 XML Fragment

```
<businessService
        serviceKey="uddi:tempuri.org:nodeservicekey"
        businessKey="uddi:tempuri.org:nodebusinesskey">
   <name xml:lang="en">UDDI Inquiry Services</name>
   <description xml:lang="en">
        Web Service supporting UDDI Inquiry APIs
   </description>
   <bindingTemplates>


<bindingTemplate
        bindingKey="uddi:tempuri.org:uddinodebindingkey_inquiry_v1"
        serviceKey="uddi:tempuri.org:inquiryservicekey">
   <description xml:lang="en">
        This binding supports the UDDI Programmer's API Specification for inquiry
   </description>
   <accessPoint useType="endPoint">
        http://tempuri.org/uddi/inquire.asmx
   </accessPoint>
   <tModelInstanceDetails>
        <tModelInstanceInfo
                tModelKey="uddi:4cd7e4bc-648b-426d-9936-443eaac8ae23">
           <description xml:lang="en">
                This access point supports the UDDI Version 1.0 Programmer's API
                Specification for inquiry
           </description>
        </tModelInstanceInfo>
   </tModelInstanceDetails>
</bindingTemplate>


<bindingTemplate
        bindingKey="uddi:tempuri.org:uddinodebindingkey_inquiry_v2"
        serviceKey="uddi:tempuri.org:inquiryservicekey">
   <description xml:lang="en">
        This binding supports the UDDI Programmer's API Specification for inquiry
   </description>
   <accessPoint useType="endPoint">
        http://tempuri.org/uddi/inquire_v2.asmx
   </accessPoint>
   <tModelInstanceDetails>
        <tModelInstanceInfo
                tModelKey="uddi:ac104dcc-d623-452f-88a7-f8acd94d9b2b">
           <description xml:lang="en">
```

```
                This access point supports the UDDI Version 2.0 Programmer's API
                Specification for inquiry
            </description>
        </tModelInstanceInfo>
    </tModelInstanceDetails>
</bindingTemplate>


<bindingTemplate
        bindingKey="uddi:tempuri.org:uddinodebindingkey_inquiry_v3"
        serviceKey="uddi:tempuri.org:inquiryservicekey">
    <description xml:lang="en">
        This binding supports the UDDI Programmer's API Specification for inquiry
    </description>
    <accessPoint useType="endPoint">
        http://tempuri.org/uddi/inquire_v3.asmx
    </accessPoint>
    <tModelInstanceDetails>
        <tModelInstanceInfo tModelKey="uddi:uddi.org:v3_inquiry">
            <description xml:lang="en">
                This access point supports the UDDI Version 3.0 Programmer's API
                Specification for inquiry
            </description>
            <instanceDetails>
                <overviewDoc>
                    <description xml:lang="en">
                        This overviewURL provides policy information about
                        inquiry.
                        </description>
                    <overviewURL useType="text">
                        http://tempuri.org/uddi/inquiry_policy.xml
                    </overviewURL>
                <instanceParms>
                    <![CDATA[
                        <?xml version="1.0" encoding="utf-8" ?>
                            <UDDIinstanceParmsContainer
                                    xmlns="urn:uddi-org:policy_v3_instanceParms">
                                <defaultSortOrder>binarySort</defaultSortOrder>
                                <authInfoUse>ignored</authInfoUse>
                            </UDDIinstanceParmsContainer>
                    ]]>
                </instanceParms>
                </overviewDoc>
            </instanceDetails>
        </tModelInstanceInfo>
    </tModelInstanceDetails>
</bindingTemplate>


    </bindingTemplates>
</businessService>
```

K.5.2 Explanation

Similar to the Publish API modeling, the Inquiry API modeling creates three bindingTemplates with three new bindingKeys. Each bindingTemplate has a different accessPoint. All three accessPoints are decorated with a useType attribute of "endPoint", as there is no indirection in accessing these APIs. Each Version implements the appropriate tModel signifying the contract necessary to invoke that API. The Version 1 and Version 2 tModels use UUIDs, whereas the Version 3 tModel, **uddi:uddi.org:v3_inquiry,** has a domainKey.

Note that the Version 3 tModel has an instanceDetails structure with an overviewDoc that leads to a policy file. Such a policy file might contain information specific to that API.

Also, there is an XML fragment nested with a "block escape" of the instanceParms. This XML fragment models the authInfo use policy for this API set as well as the default sort order.

L Appendix L: Glossary of Terms

ABNF: See Augmented Backus-Naur Form

<adjective> tModel: Any tModel that represents a concept of the type named by <adjective>. Hence, category system tModel, specification tModel, etc.

Affiliation: A collection of registries whose policies make copying data among the registries safe and easy to do.

API: Application **P**rogramming **I**nterface. (pl.: APIs.) The interface to any of the SOAP-based Web services defined in the UDDI Specification. Most are offered by UDDI nodes and invoked by clients, but validate_values, get_allValidValues, and notify_subscriptionListener are offered by clients of UDDI and invoked by a UDDI node.

API Set: Any of the collections of related UDDI APIs represented by a single specification tModel. There are 9 API sets in UDDI v3. They are inquiry, publication, subscription, custody transfer, security, subscription listener, replication, value set validation, and value set data API sets.

Appendix: Any of the lettered top-level sections of the UDDI v3 Specification.
E.g., Appendix C *Supporting Subscribers.*

Assert a relationship: To publish a publisherAssertion containing an appropriate keyedReference, toKey and fromKey, to indicate that one businessEntity is associated with another in the way indicated by the specified keyedReference. See also, Complete a relationship.

Augmented Backus-Naur Form: The formal syntax notation defined in RFC 2234. See: http://www.ietf.org/rfc/rfc2234.txt.

Authentication: the process of verifying an identity claimed by or for a system entity. See http://www.ietf.org/rfc/rfc2828.txt.

Authorization: a right or a permission that is granted to a system entity to access a system resource. See http://www.ietf.org/rfc/rfc2828.txt.

Best practice: A non-normative document accompanying a UDDI specification that provides guidance on how to use UDDI registries. Best Practices not only represent the UDDI Working Group's view on some UDDI-related topic but also represent well established practice. Cf., "Technical note".

Business: The people or organizations that are described in UDDI with a businessEntity. While quite often these are, in fact, businesses in the usual sense of the word, they need not be. For example, the "businesses" in a registry internal to a business might well be internal organizations.

Canonical: A tModel that is normative with respect to the UDDI specification. Every canonical tModel is described in Chapter 11 *Utility tModels and conventions*.

Categorize: To tag an entity with a category by placing an appropriate keyedReference or keyedReferenceGroup in the categoryBag of the entity. All entities categorized with the same value in the same category system are said to belong to the category named. E.g., in the NAICS category system, the category 481212 means "Non-scheduled chartered freight air transportation". Every businessEntity categorized with the NAICS value 481212 belongs to the NAICS *Non-scheduled chartered freight air transportation* category.

Category: A value **representing a defined division of classification** from a specified category system.

Category group system: A tModel that represents a group of category systems intended to be used to categorize entities in which it is referenced with groups of categories. Category group systems are referenced in keyedReferenceGroups and its category systems are referenced in keyedReferences in contained keyedReferences.

Category system: Any value set intended to be used to categorize the entities in which it is referenced. The documentation for a category system should describe the characteristics that mark the divisions, or categories, in the system. Each category in a category system typically represents a means of grouping distinct entities with similar characteristics.

Chapter: Any of the top-level, numbered sections of the UDDI v3 Specification.
E.g., Chapter 6 *Node Operation.*

Checked: Of or pertaining to a value set whose use is subject to validation before an entity that refers to it is published. Checked value sets may be internally hosted or externally hosted. Cf. "Unchecked".

Client: Any person who invokes one or more of the APIs covered in Chapter 5 *UDDI Programmer APIs*. Note that clients are clients of a node, not a registry. If the same person invokes APIs at two different nodes of a registry, two clients are involved. Identity of persons when they are established at all are node specific. The relationship of identities, if any, across the registry is a matter of node and registry policy.

Complete a relationship: To publish the second of two matching publisherAssertion entities. In the case in which the publisher owns referred-to businessEntity structures, asserting the relationship also completes it; no second publisherAssertion is required. Completed relationships are visible to inquirers; relationships that have not been completed are visible only to the owners of the businessEntity structures involved. See also Assert a relationship.

Container: An element in one of the UDDI API schemas that contains one other single, recurring child element. The collection of all child elements in a corresponding XML document is called a list.

Core data structures: Any of the data structures businessEntity, businessService, bindingTemplate, tModel. The core data structures are all entities.

Custody: Each entity in UDDI is said to be in the custody of the node at which it was published. That node is said to be the custodial node for the entity.

Data Integrity: the property that data has not been changed, destroyed, or lost in an unauthorized or accidental manner. See http://www.ietf.org/rfc/rfc2828.txt.

Data Confidentiality: the property that information is not made available or disclosed to unauthorized individuals, entities, or processes. See http://www.ietf.org/rfc/rfc2828.txt.

Data model: A mapping of the contents of an information model into a form that is specific to a particular type of data store or repository. UDDI specifies an information model but not a data model. Implementations are free to choose any data model they find convenient.

Delegated policy: In UDDI, a policy decision that the registry has chosen to allow a node to make. In the context of RFC 3198, the registry delegates the policy to the node.

Derived key generator: More fully, "derived key generator tModel". In the UDDI keying scheme, a key generator with a publisher-assigned key whose value is in the partition of a key generator owned by the publisher.

Digital signature: Any of the signature elements found in the optional signatures element of each UDDI entity.

Domain key generator: More fully, "domain key generator tModel". In the UDDI keying scheme, any key generator with a key of the form

"uddi:" domain ":keygenerator"

Where domain is a DNS domain – e.g., "example.com".

Element: An XML element. See http://www.w3.org/TR/REC-xml - elemdecls.

Entity: Any of the data structures that have a corresponding save_xx API. I.e., The core data structures plus publisherAssertion and subscription

Exporter: The behavior of an inquirer associated with a given registry that publishes copied (and, perhaps, modified) data into some other registry.

Find qualifier: Any of the modifiers that may be specified in the findQualifiers element of the find_xx APIs to modify the APIs' behavior. The find qualifiers are discussed in Section 5.1.4 *Find qualifiers* of the UDDI v3 Specification

Hosting redirector: An instance of a Web service whose service type is defined by the tModel uddi-org:hostingRedirector. Hosting redirector services provide a level of indirection in the retrieval of bindingTemplates. See Section B.1.4 *Using the "hostingRedirector" value* and Section 11.5.6 *UDDI Hosting Redirector Specification* for more information.

HTTP: Hyper **T**ext **T**ransfer **P**rotocol. The protocol used by the World Wide Web as defined in RFC 2068. See http://www.ietf.org/rfc/rfc2068.txt.

Identifier: A value **representing the distinct identity of the entity** from a specified identifier system.

Identifier system: Any value set intended to be used to identify the entities in which it is referenced. The documentation for an identifier system should describe the distinct characteristics of an entity for each value, or identifier, in the system. Each identifier in an identifier system typically represents a unique entity or entities that are considered equivalent.

Identify$_1$: (Concerning UDDI entities) To tag an entity with an identifier by placing an appropriate keyedReference in the identifierBag of the entity. E.g., Microsoft Business Solutions Group has the Thomas Register Supplier ID 43038249. Placing a keyedReference with this keyValue and referring to the Thomas Register Supplier ID identifier system into the identifierBag of a businessEntity identifies the businessEntity as being that of the Microsoft Business Solutions Group.

Identify$_2$: (Concerning security) To present an identifier to a system so that the system can recognize a system entity and distinguish it from other entities. (See http://www.ietf.org/rfc/rfc2828.txt)

Implementation$_1$: (of a UDDI API set) A running instance of a UDDI API set.

Implementation$_2$: (of a node) The collection consisting of the implementations of all of the API sets constituting a node.

Implementer: Any person who implements a UDDI node.

Importer: The behavior of a publisher associated with a given registry that copies (and, perhaps, modifies) data stored in one or more other registries.

Information model: An abstraction and representation of the UDDI entities in a managed environment, their properties and attributes, and the way they relate to each other. An information model is independent of any specific data model it may be mapped to. Chapters 1 through 8 of this document define the UDDI v3 information model. This includes the structure of the data, the behavior of all API sets as well as the recommended means of transport and encapsulation of the API sets. UDDI does not define a data model.

Inquirer: Any client who uses the inquiry API set.

Internally hosted: Of or pertaining to checked value sets whose use is validated entirely by the node at which the publish operation takes place. Cf., "Provider-hosted".

Internally revised: Of or pertaining to internally hosted value sets, the values for which are refreshed from time to time in some way that is entirely internal to the registry.

Internationalization: The process of generalizing computer systems so that they can handle a variety of linguistic and cultural conventions[61].

Key: Any of the unique tokens used to identify and refer to an entity stored in a registry. Keys in UDDI v3 are Uniform Resource Identifiers (URIs). See also UDDI keying scheme.

Key generator: More fully, "key generator tModel". In the UDDI keying scheme, a tModel whose key ends with ":keygenerator". Ownership of a key generator tModel represents the authority to propose keys that, so long as they are of the appropriate form, will meet the registry's policy for publisher-assigned keys. The form of the keys that is appropriate depends on the key of the tModel; the appropriate form for two different key generators is never the same.

KSS: Key Specific String

List: The collection of all elements that occur in a container element

Multi-node registry: A registry consisting of more that one node. Typically in a multi-node registry, each node has its own copy of the entities stored in the registry. In such cases the copies are typically kept synchronized using an implementation of the replication API set at each node.

Namespace: A collection of distinct names represented as strings of characters. Usually the names in a namespace are constructed according to a set of rules given by the definition of the namespace. URIs of various kinds are commonly used to construct the names in namespaces. For example, the namespace for UDDI keys in the UDDI keying scheme consists of the URIs in the "uddi" scheme.

Node: A collection of Web services, each of which implements the APIs in a UDDI API set, and that are managed according to a common set of policies. Typically, a node consists of at least an implementation of the Inquiry, the Publication, and the Custody and Ownership Transfer API sets; often a node will implement additional API sets such as Subscription and Replication.

Node Business Entity: A businessEntity in a registry that is categorized using the uddi-org:operators category system. At a minimum, each Node Business Entity describes the Web services that constitute the node.

Non-normative: Information included in the UDDI v3 Specification that is advisory or explanatory and does not constitute required aspects of the specification.

Normative: Specification information that is intrinsic to the UDDI v3 Specification and must be adhered to.

Operator: The role of a person who sets node policy and runs a node. There is exactly one operator for a given node.

Overview document: A document providing the technical definition of the concept a given tModel represents. The overview documents for a given tModel may be retrieved using the URLs contained within the overviewDoc structure of the tModel.

Owner: The publisher who has the authority to change a given entity.

[61] Martin O'Donnell, Sandra. Programming for the World. Prentice Hall, Englewood Cliffs, NJ. 1994.

Partition: In the UDDI keying scheme, the appropriate form of publisher-assigned keys for a given tModel. The exact definition for the partition of a key generator is given in Section 5.2.2.1 *Key generator keys and their partitions*.

Person: Someone – for example, an actual individual, an organization, or a job role – who interacts with a UDDI registry in some way.

Policy: Certain behaviors, as identified in Chapter 9 *Policy,* that are permitted to vary from registry to registry or even from node to node within a registry. The choice of specific behavior made by a registry or node is its policy with respect to the variable behavior. As described in Chapter 9, each registry and node describes the policies it adheres to.

Policy abstractions: Broad definitions of high level information management policies. Policy can be represented at different levels, ranging from business goals to device specific configuration parameters. Translation between different levels of "abstraction" may require information other than policy, such as network and host parameter configuration and capabilities. In UDDI there are often ways to model the policies as a tModel. In other cases the policy may be documented.

Policy decision: A process perspective that deals with the evaluation of a policy rule's conditions. When an instance of UDDI is implemented, choices are made about the support for the UDDI APIs. There are points at which the implementation is responsible for a determination about whether or not the request is processed based on the rule's specified for that node.. Because UDDI can be implemented in a variety of ways, these actions are considered policy decisions and the implementation must document how these decisions are made within this implementation of UDDI.

Policy decision point: A logical entity that makes policy decisions for itself or for other network elements that request such decisions. See http://www.ietf.org/rfc/rfc2753.txt. The UDDI registry and its delegates make policy decisions.

Policy enforcement: The execution of a policy decision.

Policy enforcement point: A logical entity that enforces policy decisions. See http://www.ietf.org/rfc/rfc2753.txt. The UDDI API implementations enforce policy decisions.

Policy goals: The business objectives or desired state intended to be maintained by a policy system. At the highest level of abstraction of policy, these goals are most directly described in business rather than technical terms.

Policy rule: A basic building block of a policy-based system. It is the binding of a set of actions to a set of conditions where the conditions are evaluated to determine whether the actions are performed.

Postal address system: Any value set intended to be used to define the meaning of addressLine structures.

Project (verb): To establish the existence of a service projection. E.g., "Business A *projects* the View Catalog business service belonging to business B in its businessEntity."

Protocol tModel: A tModel representing a protocol such as the standard fax protocol. Cf., "Specification tModel" and "Transport tModel".

Provider-revised: Of or pertaining to internally hosted value sets, the values for which are revised from time to time by the value set publisher. Publisher-revised value sets have a get_allValidValues Web service associated with them from which nodes retrieve the revised value set values.

Provider-hosted: Of or pertaining to checked value sets whose use is validated using a validate_values Web service.

Publisher: Any client who uses the publish API set.

Publish: The act of placing one or more entities in a registry by invoking one of the publication APIs.

Publisher-assigned key: Any key created by a publisher (as opposed to one created by a node.) A publisher may *propose* a key for a new entity at the time it is first published. If the proposed key meets the registry's policy for publisher-assigned keys and the publish operation succeeds, the key proposed by the publisher becomes a publisher-assigned key.

Registry: One or more UDDI nodes may be combined to form a UDDI Registry. The nodes in a UDDI registry collectively manage a particular set of UDDI data. This data is distinguished by the visible behavior associated with the entities contained in it. A UDDI Registry has these defining characteristics: a registry is comprised of one or more UDDI nodes; the nodes of a registry collectively manage a well-defined set of UDDI data. Typically, this is supported by the use of UDDI replication between the nodes in the registry which reside on different systems; and a registry MUST make a policy decision for each policy decision point. It MAY choose to delegate policy decisions to nodes. See Chapter 9 *Policy* for details. The physical realization of a UDDI Registry is not mandated by this specification.

Registry administrator: The role of the person who sets the policies for a registry. There is exactly one registry administrator for every registry. Typically, a registry administrator is an organization consisting of the operators of the registry's nodes.

Registry-assigned key: If no key is proposed for an entity at the time it is first published, the registry assigns a key. Such keys are called registry-assigned keys.

Relationship: An association between one businessEntity and another established by completing a relationship. See also Assert a relationship and Complete a relationship

Relationship type: A value from a specified relationship type system.

Relationship type system: Any value set intended to be used to define the relationship types that are used in relationships between businessEntities.

Romanization: The practice in some cultures of transliterating words, particularly names, from their usual written form into Roman letters so that people unfamiliar with the usual form can (make some sort of attempt to) pronounce them. In Japan, for example, it is common practice to Romanize personal names and place names for the benefit of people who cannot read Japanese.

Root key generator: More fully, "root key generator tModel". In the UDDI keying scheme either a "domain key generator" or a "uuid key generator".

Section: Any of the titled parts of a chapter or appendix.
E.g., Section C.2 *Using subscription*.

Service projection: A projected businessService is made a part of a businessEntity by reference as opposed to by containment. Every businessService is "contained" in exactly one businessEntity. The businessEntity the businessService is contained in is the one whose businessKey is specified in the businessKey attribute of the businessService in question. Therefore, if the businessKey found in a businessService is not the same as the businessKey of the businessEntity in which it is found, the businessService is a service projection. See also Project.

Service type: The type of a Web service as defined by its technical fingerprint.

Short name: Any of the abbreviated names for common find qualifiers as defined in Section 5.1.4 *Find Qualifiers*.

signatureComponent tModel: A tModel representing a component of a Web service specification. Some specifications, such as the RosettaNet are deliberately broken into pieces meant to be composed together to form a complete description of a Web service a signatureComponent tModel is used to represent each of these pieces.

SMTP: Simple **M**ail **T**ransport **P**rotocol. The mail transport mechanism commonly used on the Internet. Defined in RFC 2821. See: http://www.ietf.org/rfc/rfc2821.txt.

SOAP: Simple **O**bject **A**ccess **P**rotocol Version 1.1. The most common Web service protocol. Defined in World Wide Web Consortium Note. See: http://www.w3.org/TR/SOAP/.

soapSpec tModel: An xmlSpec tModel whose specification expresses its data using SOAP 1.1.

Sort order: Any of the find qualifiers that specify the base algorithm to be used to order the results returned by the find_xx APIs

Specification tModel: A tModel representing the specification for a Web service type. Cf. "Protocol tModel" and "Transport tModel".

Structure: Any of the UDDI elements that are permitted by the schemas to have elements contained within them.

SSL: Secure **S**ockets **L**ayer Version 3.0. A commonly used network protocol as defined in http://home.netscape.com/eng/ssl3/index.html.

Subscriber: Any client who uses the subscription API set.

Technical fingerprint: The collection of tModel references found in the tModelInstanceDetails of a bindingTemplate. This collection, taken without regard to order, defines the type of the Web service described by the bindingTemplate.

Technical Note: A non-normative document accompanying the UDDI v3 Specification that provides guidance on how to use UDDI registries. While technical notes represent the UDDI Working Group's view on some UDDI-related topic, they are often prospective and need not document existing practice. Cf. Best Practice.

tModel: The type of entity in UDDI that is used to represent concepts.

Transport tModel: A tModel representing a transport mechanism such as HTTP. Typically used in a technical fingerprint, if required by the specification tModel, to specify which transport mechanism a given Web service instance uses.

UDDI: Universal **D**escription, **D**iscovery and **I**ntegration. The subject of this specification.

UDDI Business Registry: A publicly available UDDI registry. The Web services offered by the UDDI Business Registry's nodes can be found at http://uddi.org/register.html (publication) and http://uddi.org/find.html (inquiry).

UDDI keying scheme: The URI scheme "uddi" together with the set of registry policies that makes copying data among registries safe and easy to do. The keying scheme is described in Chapter 4 of the UDDI v3 Specification

UDDI schema: Any of the documents written in the XML Schema Definition Language that accompany this specification and are a part of it. See Chapter 2 *UDDI Schemas*.

UDDI v3 Specification, The: The main textual document that specifies UDDI version 3.

UDDI Version 2: The prior version of this specification. See http://uddi.org/pubs/ProgrammersAPI_v2.pdf.

Unicode: The character set defined by the Unicode Consortium. See www.unicode.org. UDDI data is express using the Unicode character set.

UTF-16: An encoding scheme used by UDDI to express Unicode characters. See: http://www.ietf.org/rfc/rfc2781.txt.

UTF-8: An encoding scheme used by UDDI to express Unicode characters. See http://www.ietf.org/rfc/rfc2279.txt.

uuid key generator: More fully, "uuid key generator tModel". In the UDDI keying scheme, a key generator with a key of the form

"uddi:" uuid ":keygenerator"

Where uuid is a Universally Unique Identifier.

Unchecked: Of or pertaining to value sets whose use in not subject to validation. Cf., "Checked".

URI: A Uniform Resource Identifier as represented by the XML Schema data type anyURI. When, as it occasionally is, the term is used more narrowly, the context makes this clear.

UUID: A really big integer, generated according to certain rules that allow them to be generated on independent computers in such a way that the same one is never generated twice. UUIDs are conventionally expressed as specially formatted hexadecimal.

Value set: Any category system, identifier system, relationship system or postal address system.

Value set provider: Any publisher who publishes a checked value set tModel. Typically a value set provider also provides a get_allValidValues or a validate_values Web service or both.

Web service: Any service capable of being described by a UDDI bindingTemplate. Typically Web services are used for machine-to-machine communication and typically, they share much of the technology that underlies the World Wide Web, such as TCP/IP, HTTP, and XML.

Web service type: Informally, the type of a Web service instance is defined by the external behaviors it exhibits. Typically these are defined by the specifications, protocols, and transports to which it adheres. Formally in UDDI, the type of a Web service is the technical fingerprint of the bindingTemplate that describes it.

WSDL: Web **S**ervices **D**escription **L**anguage Version 1.0. An XML vocabulary and set of conventions used to describe the technical details of a Web service, particularly of a SOAP-based Web service. See: http://www.w3.org/TR/wsdl.

wsdlSpec tModel: An xmlSpec tModel whose description is expressed in WSDL and which follows the UDDI Best Practice for the combined use of UDDI and WSDL.

XML: e**X**tensible **M**ark-up **L**anguage. A platform and implementation-neutral way to describe data. See: http://www.w3c.org/TR/2000/REC-xml-20001006

xmlSpec tModel: A specification tModel whose specification expresses its data using XML.

M Appendix M: Acknowledgements

This specification was developed as a result of joint work of many individuals from the OASIS UDDI Specification TC and the former UDDI.org Working Group. We would like to acknowledge the efforts and contributions of the following individuals:

Former OASIS UDDI Specification TC members:

Bob Atkinson, Microsoft
Toufic Boubez, Layer Seven Technologies
Maud Cahuzac, France Telecom
Ugo Corda, SeeBeyond Technology
Alexandru Czimbor, OSS Nokalva
Patrick Felsted, Novell
Shishir Garg, France Telecom
Rajul Gupta, OSS Nokalva
Brad Henry
Karsten Januszewski, Microsoft
Aikichi Kawai, NTT USA
Keisuke Kibakura, Fujitsu
Eric Lee, Microsoft
Sam Lee, Oracle
Anne Thomas Manes, Individual Member
Alok Srivastava, Oracle
Paul Thorpe, OSS Nokalva
Alessandro Triglia, OSS Nokalva
Max Voskob, Individual Member
George Zagelow, IBM

UDDI Working Group members (at the time the UDDI Version 3.0 specification was published):

Selim Aissi, Intel
David Ehnebuske, IBM (Editor)
Tom Gaskins, HP
Tom Glover, IBM
Christian Hansen, SAP
Thomas Hardjono, VeriSign
Richard Harrah, HP
Maryann Hondo, IBM (Editor)
Yin Leng Husband, HP (Editor)
Karsten Januszewski, Microsoft (Editor)
Keisuke Kibakura, Fujitsu
Sam Lee, Oracle (Editor)
Seán MacRoibeáird, Sun
Barbara McKee, IBM (Editor)
Joel Munter, Intel (Editor)
Ed Mooney, Sun
Andrew Nielsen, HP
Christian R. Thomas, Intel
Johannes Viegener, SAP

N Appendix N: Notices