Liberty ID-WSF Data Services Template

Version: 2.1

Editors:
Sampo Kellomäki, Symlabs, Inc.
Jukka Kainulainen, Nokia Corp.

Contributors:
Robert Aarts, Hewlett-Packard
Rajeev Angal, Sun Microsystems, Inc.
Conor Cahill, America Online, Inc.
Carolina Canales-Valenzuela, Ericsson
Darryl Champagne, IEEE-ISTO
Andy Feng, America Online, Inc.
Gael Gourmelen, France Télécom
Jeff Hodges, NeuStar, Inc.
Lena Kannappan, France Télécom
John Kemp, Nokia Corporation
Rob Lockhart, IEEE-ISTO
Paul Madsen, NTT
Aravindan Ranganathan, Sun Microsystems, Inc.
Matti Saarenpää, Nokia Corporation
Jonathan Sergent, Sun Microsystems, Inc.
Lakshmanan Suryanarayanan, America Online, Inc
Greg Whitehead, Hewlett-Packard

Abstract:

The Data Services Template provides protocols, schema and processing rules for the query, creation, deletion, and modification of data objects and their attributes exposed by a data service using the Liberty Identity Web Services Framework (ID-WSF). Some guidelines and common XML attributes and data types for data services are defined.

Filename: liberty-idwsf-dst-v2.1.pdf

Notice

This document has been prepared by Sponsors of the Liberty Alliance. Permission is hereby granted to use the document solely for the purpose of implementing the Specification. No rights are granted to prepare derivative works of this Specification. Entities seeking permission to reproduce portions of this document for other uses must contact the Liberty Alliance to determine whether an appropriate license for such use is available.

Implementation of certain elements of this document may require licenses under third party intellectual property rights, including without limitation, patent rights. The Sponsors of and any other contributors to the Specification are not and shall not be held responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights. **This Specification is provided "AS IS", and no participant in the Liberty Alliance makes any warranty of any kind, express or implied, including any implied warranties of merchantability, non-infringement of third party intellectual property rights, and fitness for a particular purpose.** Implementers of this Specification are advised to review the Liberty Alliance Project's website (http://www.projectliberty.org/) for information concerning any Necessary Claims Disclosure Notices that have been received by the Liberty Alliance Management Board.

Liberty Alliance Project
Licensing Administrator
c/o IEEE-ISTO
445 Hoes Lane
Piscataway, NJ 08855-1331, USA
info@projectliberty.org

Contents

1. Introduction

This specification provides protocols for the creation, query, modification, and deletion (a.k.a. "CRUD") of data attributes, exposed by a data service, related to a Principal. Some guidelines, common XML attributes and data types are defined for data services.

This specification does not give a strict definition as to which services are data services and which are not, i.e., to which services this specification is targeted. A data service, as considered by this specification, is a web service that supports the storage and update of specific data attributes regarding a Principal. A data service might also expose dynamic data attributes regarding a Principal. Those dynamic attributes may not be stored by an external entity, but the service knows or can dynamically generate their values.

An example of a data service would be a service that hosts and exposes a Principal's profile information (such as name, address and phone number). An example of a data service exposing dynamic attributes is a geolocation service.

The data services using this specification can also support other protocols than those specified here. They are not restricted to support just querying and modifying data attributes, but they can also support actions (e.g., making reservations). Also some services might support only querying data without supporting modifications and in some cases there could be services supporting only modifications without supporting querying, i.e., other parties are allowed to give new data, but not query existing. The specification provides many features and data services must choose which features to use and how to use them.

This specification has three main parts. First some common attributes, guidelines and type definitions to be used by different data services are defined and the XML schema for those is provided. Second, the methods of accessing the data are provided, including an XML schema for the Data Services Template (DST) protocols. Finally, a checklist is given for writing services on top of the DST.

1.1. Notation

When capitalized, the key words "MUST," "MUST NOT," "REQUIRED," "SHALL," "SHALL NOT," "SHOULD," "SHOULD NOT," "RECOMMENDED," "MAY," and "OPTIONAL" in this specification are to be interpreted as described in [RFC2119]. When these words are not capitalized, they are meant in their natural-language sense.

This specification uses the following typographical conventions in text: **<Element>**, **<ns:ForeignElement>**, `attribute`, `DataType`, `OtherCode`.

For readability, when an XML Schema type is specified to be `xs:boolean`, this document discusses the values as "`true`" and "`false`" rather than the "1" and "0" which will exist in the document instances.

Definitions for Liberty-specific terms can be found in [LibertyGlossary].

1.2. Liberty Considerations

This specification contains enumerations of values that are centrally administered by the Liberty Alliance Project. Although this document may contain an initial enumeration of approved values, implementers of the specification MUST implement the list of values whose location is currently specified in [LibertyReg] according to any relevant processing rules in both this specification and [LibertyReg].

1.3. Namespaces

The namespaces described in table 1 are used.

Table 1. Normatively referenced XML namespaces

Prefix	URI	Description
dst:	urn:liberty:dst:2006-08	Target namespace of DST utility schema.
dstref:	urn:liberty:dst:2006-08:ref	Target namespace of DST reference model.
xml:	http://www.w3.org/XML/1998/namespace	W3C XML [XML]
xs:	http://www.w3.org/2001/XMLSchema	W3C XML Schema Definition Language [Schema1-2]
ds:	urn:liberty:disco:2006-08	Liberty ID-WSF Discovery Service [Liberty-Disco]
lu:	urn:liberty:util:2006-08 Li	erty Utility schema

1.4. Applying DST to Define Services

In order to define a service the service specification is expected to reference DST for common processing rules and utility schema. Where common definitions are not appropriate, the service specification is expected to

a. Answer every question specified in the check list, see Section 10

b. Waive inappropriate processing rules

c. Define additional processing rules

d. Alter DST SHOULD statements to either MUST or MUST NOT if appropriate

e. Define service schema in terms of DST utility schema. It is RECOMMENDED that the schema mimic the Reference Model, see Section 11.1, as appropriate. The service schema is likely to define at least `AppDataType` and possibly other service specific aspects.

1.5. Applying the DST Reference Model

The DST reference model, see Section 11.1, depicts a prototypical service schema. The dstref: namespace would be substituted by the service specific namespace. Since the service is fully defined by its own independent schema, it is free to redefine all aspects as it sees fit. However, to promote common approach to data services, it is RECOMMENDED that the service follow this reference model wherever there is no specific reason to diverge from it.

In particular, when this document specifies processing rules, the method names, such as <**Create**>, <**Query**>, etc., specified by the reference model are used. If service schema chooses other method names, it needs to specify correspondence to reference model method names so that applicable processing rules can be determined.

```
<xs:element name="Create" type="dstref:CreateType"/>
<xs:element name="CreateResponse" type="dstref:CreateResponseType"/>
<xs:element name="Query" type="dstref:QueryType"/>
<xs:element name="QueryResponse" type="dstref:QueryResponseType"/>
<xs:element name="Modify" type="dstref:ModifyType"/>
<xs:element name="ModifyResponse" type="dstref:ModifyResponseType"/>
<xs:element name="Delete" type="dstref:DeleteType"/>
<xs:element name="DeleteResponse" type="dstref:DeleteResponseType"/>
```

Figure 1. Reference Definitions of Methods

The reference model provides dummy definitions of some important extension points. Typical service schema will provide its own definitions for these.

```
<xs:complexType name="SelectType">
  <xs:simpleContent>
    <xs:extension base="xs:string"/>
  </xs:simpleContent>
</xs:complexType>
<xs:complexType name="TestOpType">
  <xs:simpleContent>
    <xs:extension base="xs:string"/>
  </xs:simpleContent>
</xs:complexType>
<xs:complexType name="SortType">
  <xs:simpleContent>
    <xs:extension base="xs:string"/>
  </xs:simpleContent>
</xs:complexType>
<xs:complexType name="AppDataType">
  <xs:simpleContent>
    <xs:extension base="xs:string"/>
  </xs:simpleContent>
</xs:complexType>
```

Figure 2. DST Parameterization Points

2. Data Model

A data service provides access to the data. The data consists of one or more objects and there can be multiple objects of same type. For each different type of a data service the supported objects must be specified. One type of data service might support only one object, another might support multiple objects of same type and a third might support multiple types of objects and multiple instances of objects of the same type. For each service type an XML schema must be specified. There can also be multiple XML schemata for one service type as different data objects might be in different schemata. The XML schema for a service type defines the data that the service type can host and the structure of this data. See [LibertyDisco] for more information about service types.

A data object has a root element which contains data in subelements. The name of this root element is used as the object type identifier. Individual objects can be accessed by defining the object type and selecting from the objects of that type the right one. Selecting can be done using an identifier, which is unique among those objects, using some data values object contains or using some service type specific parameters, which give enough information to a service so that it can calculate, what data the requestor wants to access. Individual data elements inside objects can also be accessed separately, e.g., from a contact card the name can be queried separately. The specification for each service type defines in details, how the selecting is done. This document gives common rules, but the actual selection mechanism is specified in the service specifications.

The data may be stored in implementation specific ways, but will be exposed by the service using the XML schema specified both in this document, and that of the defined service type. This also means that the XML document defined by the schema is a conceptual XML document. Depending upon the implementation, there may be no XML document that matches the complete conceptual document. The internal storage of the data is separate and distinct from the document published through this model.

The schemata for different service types may have common characteristics. This section describes the commonalities specified by the Data Services Template, provides schema for common XML attributes and data types, and also gives some guidelines.

2.1. Guidelines for Schemata

The schemata of different data services SHOULD follow guidelines defined here. The purpose of these guidelines is to make the use of the Data Services Template easier when defining and implementing services.

1. Each data attribute regarding the Principal SHOULD be defined as an XML element of a suitable type.

2. XML attributes SHOULD be used only to qualify the data attribute defined as XML elements and not contain the actual data values related to the Principal.

3. An XML element SHOULD either contain other XML elements or actual data value. An XML element SHOULD NOT have `mixed` content, i.e., both a value and sub-elements. Also complex types `all` and `choice` SHOULD NOT be used.

4. Once a data attribute has been published in a specification for a service type, its syntax and semantics MUST not change. If evolution in syntax or semantics is needed, any new version of a data attribute MUST be assigned a different name, effectively creating a new attribute with new semantics so that it does not conflict with the original attribute definition.

5. All elements MUST be defined as global elements, when they can be requested individually. When elements with complex type are defined, references to global elements are used. The reason for this guideline is that the XML Schema for a service does not only define the syntax of the data supported by the service but also the transfer syntax. In many cases it should be possible to query and modify individual elements.

6. The type definitions provided by the XML Schema SHOULD be used, when they cover the requirements.

2.2. Extending a Service

A service, defined by its specification and schema, MAY be extended in different ways. What types of extensions are supported in practice MUST be specified individually by each service specification, or agreed locally between the WSC and WSP.

1. An implementation MAY add new elements and XML attributes to an already specified object or it may add totally new objects. The new data MUST use its own namespace until it is added to the official service specification and schema of the service type.

2. When new features for a service are specified (e.g., new elements), new keywords SHOULD be specified for indicating the new features using the **<Option>** element (see [LibertyDisco] for more information).

3. New values for enumerators MAY be specified subsequent to the release of a specification document for a specific service type. The specification for a service type MUST specify the authority for registering new official enumerators (whether that authority is the specification itself, or some external authority). For specification done by Liberty Alliance, see [LibertyReg].

4. Elements defined in the XML schema for a service type MAY contain an **<xs:any>** element to support arbitrary schema extension. When the **<xs:any>** elements are in the schema, an implementation MAY support this type of extension, but is not required to. The **<xs:any>** elements SHOULD always be put inside **<Extension>** elements. If an implementation does support this type of schema extension, then it MAY register the urn:liberty:dst:can:extend discovery option keyword. When a service holds new data, which is not defined in the schema for the service type but is stored using this kind of support for extensions, it MAY register the urn:liberty:dst:extend discovery option keyword.

The <Extension> Element

All messages have an **<Extension>** element for services which need more parameters. The **<Extension>** element SHOULD NOT be used in a message, unless its content and related processing rules have been specified for the service. If the receiving party does not support the use of the **<Extension>** element, it MUST ignore it.

2.3. Time Values and Synchronization

Some of the common XML attributes are time values. All Liberty time values have the type `dateTime`, which is built in to the W3C XML Schema Data Types specification. Liberty time values MUST be expressed in the UTC (a.k.a. GMT or the "Zulu" time) form, indicated by a "Z" immediately following the time portion of the value.

Liberty requestors and responders SHOULD NOT rely on other applications supporting time resolution finer than seconds, as implementations MAY ignore fractional second components specified in timestamp values. Implementations MUST NOT generate time instants that specify leap seconds.

The timestamps used in the DST schemata are only for the purpose of data synchronization and no assumptions should be made as to clock synchronization. As clocks might not be well synchronized, a WSC SHOULD check the general timestamps returned in response messages and compare those to its own clock. This helps a WSC to better evaluate different timestamps attached to different data items.

2.4. Common XML Attributes

The XML elements defined in the XML schemata for the services either contain data values or other XML elements. So an XML element is either a leaf element or a container. The containers MUST NOT have any other data content than other XML elements and possible qualifying XML attributes. To contrast, the *leaf* elements do not contain other XML elements. These leaf elements can be further divided into two different categories: normal and localized. The normal leaf elements typically contain a string or URI constant. The localized leaf elements contain text using a local writing system.

Both leaf and container XML elements can have service-specific XML attributes, but there are also common XML attributes supplied for use by all data services. These common XML attributes are technical attributes, which are usually created by the Web Service Provider (WSP) hosting a data service (for more details, see Section 7). These technical attributes are not mandatory for all data services, but if they are implemented, they MUST be implemented in the way described in this document. Each service should specify separately if one or more of these common XML attributes are mandatory or optional for that service. In addition to the common XML attributes, we define attribute groups containing these common XML attributes. There are three attribute groups: `commonAttributes` mainly targeted for container elements and for the leaf elements `leafAttributes` and `localizedLeafAttributes`.

```
<xs:attribute name="id" type="lu:IDType"/>
<xs:attribute name="modificationTime" type="xs:dateTime"/>
<xs:attributeGroup name="commonAttributes">
  <xs:attribute ref="dst:id" use="optional"/>
  <xs:attribute ref="dst:modificationTime" use="optional"/>
</xs:attributeGroup>
<xs:attribute name="ACC" type="xs:anyURI"/>
<xs:attribute name="ACCTime" type="xs:dateTime"/>
<xs:attribute name="modifier" type="xs:string"/>
<xs:attributeGroup name="leafAttributes">
  <xs:attributeGroup ref="dst:commonAttributes"/>
  <xs:attribute ref="dst:ACC" use="optional"/>
  <xs:attribute ref="dst:ACCTime" use="optional"/>
  <xs:attribute ref="dst:modifier" use="optional"/>
</xs:attributeGroup>
<xs:attribute name="script" type="xs:anyURI"/>
<xs:attributeGroup name="localizedLeafAttributes">
  <xs:attributeGroup ref="dst:leafAttributes"/>
  <xs:attribute ref="xml:lang" use="required"/>
  <xs:attribute ref="dst:script" use="optional"/>
</xs:attributeGroup>
<xs:attribute name="refreshOnOrAfter" type="xs:dateTime"/>
<xs:attribute name="destroyOnOrAfter" type="xs:dateTime"/>
```

Figure 3. DST Common XML Attributes

2.4.1. The `commonAttributes` XML Attribute Group

There are only two common XML attributes:

`id` (optional) The `id` is a unique identifier within a document. It can be used to refer uniquely to an element, especially when there may be several XML elements with the same name. If the schema for a data service does not provide any other means to distinguish between two XML elements and this functionality is needed, the `id` XML attribute MUST be used. It is only meant for distinguishing XML elements within the same conceptual XML document. It MUST NOT be a globally unique identifier, because that would create privacy problems. An implementation MAY set specific length restrictions on `id` XML attributes to enforce this. The value of the `id` XML attribute SHOULD stay the same when the content of the element is modified so the same value of the `id` XML attribute can be used when querying the same elements at different times. The `id` XML attribute MUST NOT be used for storing any data and it SHOULD be kept short.

`modificationTime` (optional) The `modificationTime` specifies the last time that the element was modified. Modification includes changing either the value of the element itself, or any sub-element. So the time of the modification MUST be propagated up all the way to the root element, when container elements have the `modificationTime` XML attribute. If the root element has the `modificationTime` XML attribute, it states the time of the latest modification. Note that a data service may have the `modificationTime` XML attribute used only in leaf elements or not even for those as it is optional.

2.4.2. The `leafAttributes` XML Attribute Group

This group includes the `commonAttributes` XML attribute group and defines three more XML attributes for leaf elements (XML elements not containing other XML elements):

`modifier` (optional) The `modifier` is the `ProviderID` of the service provider which last modified the data element.

`ACC` (optional) The acronym `ACC` stands for *Attribute Collection Context* which describes the context (or mechanism) used in collecting the data. This might give useful information to a requestor, such as whether any validation has been done. The `ACC` always refers to the current data values, so whenever the value of an element is changed, the value of the `ACC` must be updated to reflect the new situation. The `ACC` is of type **`anyURI`**.

The following are defined values for the `ACC` XML attribute:

urn:liberty:dst:acc:unknown This means that there has been no validation, or the values are just voluntary input from the user. The `ACC` MAY be omitted in the message exchange when it has this value, as this value is equivalent to supplying no `ACC` XML attribute at all.

urn:liberty:dst:acc:incentive There has been some incentive for user to supply correct input (such as a gift sent to the user in return for their input).

urn:liberty:dst:acc:challenge A challenge mechanism has been used to validate the collected data (e.g., an email sent to address and a reply received or an SMS message sent to a mobile phone number containing a WAP URL to be clicked to complete the data collection)

urn:liberty:dst:acc:secondarydocuments The value has been validated from secondary documents (such as the address from an electric bill).

urn:liberty:dst:acc:primarydocuments The value has been validated from primary documents (for example, the name and identification number from a passport).

Other values are allowed for `ACC`, but this specification normatively defines usage only for the values listed above.

When the `ACC` is included in the response message, the response SHOULD be signed by the service provider hosting the data service.

`ACCTime` (optional) This defines the time that the value for the `ACC` XML attribute was given. Note that this can be different from the `modificationTime`. The `ACC` contains information that may be related to the validation of the entry. Such validation might happen later than the time the entry was made, or modified. The entry can be validated more than once.

2.4.3. The `localizedLeafAttributes` XML Attribute Group

This XML attribute group includes the `leafAttributes` XML attribute group and defines two more XML attributes to support localized data. UTF-8 is capable of carrying the data in the right format, but it is difficult to access out of the XML elements having the same name the one containing the information in the right language and writing system. These XML attributes should be used when multiple languages can be used and it is important to be able to get the data in the right language and writing system.

`xml:lang` (required) This defines the language used for the value of a localized leaf element. When the **<localizedLeafAttributes>** XML attribute group is used for an element, this is a mandatory XML attribute.

`script` (optional) Sometimes the language does not define the writing system used. In such cases, this XML attribute defines the writing system in more detail. This specification defines the following values for this XML attribute: urn:liberty:dst:script:kana and urn:liberty:dst:script:kanji. See [LibertyReg] where to find additional values, if any, and how to specify more values.

2.4.4. Individual Common XML attributes

In addition to the previous XML attribute groups a couple of more common XML attributes are defined and available for services. The XML attributes in XML attribute groups can also be used individually without taking the whole attribute group into use, but the following XML attributes are assumed to be seldom used and so they are not included in any of the XML attribute groups.

`refreshOnOrAfter` A WSC may cache the information in the element and if it chooses to do so, it SHOULD refresh the data from the WSP if it attempts to use the data beyond the time specified. If the data is not refreshed (for whatever reason) a WSC MAY continue to use it. This parameter does NOT place an obligation upon the WSP to keep the value of the data static during this timespan, so it is possible (and in some cases likely) that the contents of the element will change during the specified period. WSCs that require timely data should request the most up to date data when they need it rather than caching the data.

`destroyOnOrAfter` Even if a WSC has not been able to refresh the information, it SHOULD destroy it, if the element containing the information has the XML attribute `destroyOnOrAfter` and the time specified by that attribute has come. The information most probably is so out of date that it is unusable.

2.5. Common Data Types

The type definitions provided by XML schema can not always be used directly by Liberty ID-WSF data services, as they lack the common XML attributes noted above. The DST data type schema provides types derived from the XML Schema ([XML]) data type definitions with those common XML attributes added to the type definitions. Please note that for strings there are two type definitions, one for localized elements and another for elements normalized using the Latin 1 character set. The common data type definitions are depicted in Figure 4.

```
<xs:complexType name="DSTLocalizedString">
  <xs:simpleContent>
    <xs:extension base="xs:string">
      <xs:attributeGroup ref="dst:localizedLeafAttributes"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
<xs:complexType name="DSTString">
  <xs:simpleContent>
    <xs:extension base="xs:string">
      <xs:attributeGroup ref="dst:leafAttributes"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
<xs:complexType name="DSTInteger">
  <xs:simpleContent>
    <xs:extension base="xs:integer">
      <xs:attributeGroup ref="dst:leafAttributes"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
<xs:complexType name="DSTURI">
  <xs:simpleContent>
    <xs:extension base="xs:anyURI">
      <xs:attributeGroup ref="dst:leafAttributes"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
<xs:complexType name="DSTDate">
  <xs:simpleContent>
    <xs:extension base="xs:date">
      <xs:attributeGroup ref="dst:leafAttributes"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
<xs:complexType name="DSTMonthDay">
  <xs:simpleContent>
    <xs:extension base="xs:gMonthDay">
      <xs:attributeGroup ref="dst:leafAttributes"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

Figure 4. General Data Types with DST Attributes

3. Message Interface

This specification defines number of protocols for data services. These protocols rely mainly on a request-response message exchange pattern. The only exceptions are the notification messages, which might not get any response. The messages specified in this document are carried in the SOAP body. No additional content is specified for the SOAP header in this document, but implementers of these protocols MUST follow the rules defined in [LibertySOAPBinding], including passing credentials or target ID that allows the resource to be accessed to be determined.

The following table lists the protocol elements specified by this specification (with respect to the DST reference model).

Table 2. Requests and Responses

Request by a WSC	Response by a WSP
<Create>	**<CreateResponse>**
<Delete>	**<DeleteResponse>**
<Query>	**<QueryResponse>**
<Modify>	**<ModifyResponse>**

<Create> and **<Delete>** are used to create new objects and delete existing objects. The data inside an object can be modified using **<Modify>**, this includes deleting individual data items inside an object. Whole objects or data inside objects can be queried using **<Query>**.

The messages for different protocols have common features, XML attributes and elements. These common issues are discussed in this chapter and the actual messages are specified in the following chapters. Together with common parts the related processing rules are also defined. In the text, especially in the processing rules, the *RequestElement* is used to replace the actual request element in many cases. These parts MUST be read as if instead of a *RequestElement* there would be any of the following elements: **<Create>**, **<Delete>**, **<Query>** or **<Modify>**.

The *ResponseElement* is used instead of the actual response element in many places. Those parts MUST be read as if instead of a *ResponseElement* there would be any of the following elements: **<CreateResponse>**, **<DeleteResponse>**, **<QueryResponse>** or **<ModifyResponse>**.

```
<xs:complexType name="RequestType">
  <xs:sequence>
    <xs:element ref="lu:Extension" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute ref="lu:itemID" use="optional"/>
  <xs:anyAttribute namespace="##other" processContents="lax"/>
</xs:complexType>
<xs:complexType name="DataResponseBaseType">
  <xs:complexContent>
    <xs:extension base="lu:ResponseType">
      <xs:attribute name="timeStamp" use="optional" type="xs:dateTime"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Figure 5. Commonality of Requests and Responses

3.1. Multiple Occurrences of Request or Response

If service specification permits, all request and response elements MAY occur multiple times in the message (e.g., the SOAP **<body>** if the SOAP binding is used). This mechanism can serve as a batch optimization or the service specification MAY choose to attach some transactional semantics to this construct.

3.2. Status and Fault Reporting

Two mechanism are defined to report back to the requestor whether the processing of a request was successful or not or something in between. [LibertySOAPBinding] defines the ID-* Fault message, which is used to convey processing exception. An ordinary ID-* Message carrying normal response is used to report back application statuses including normal error conditions, when an application has detected an error condition as part of the normal processing, e.g., processing according to the processing rules specified in this document.

From the Data Service Template point of view there are the following cases in which the ID-* Fault Message is used.

1. When a WSP does not recognize any *RequestElement* in the SOAP Body, it MUST return an ID-* Fault Message and use `IDStarMsgNotUnderstood` as the value of the `code` XML attribute as specified by [LibertySOAP-Binding]. This fault MAY also be applied to situations where implementation or deployment has permanently chosen not to support certain type of request (e.g., read only service).

2. In the same way, a WSC that receives an empty or malformed notification MUST return an ID-* Fault Message and use `IDStarMsgNotUnderstood` as the value of the `code` XML attribute.

3. If a WSP based on identifying the requesting party notices that the requesting party is not allowed to make any requests, it MUST return an ID-* Fault Message and use `ActionNotAuthorized` as the value of the `code` XML attribute.

4. A receiving party may also encounter an unexpected error due to which it fails to handle the message body. In that case it MUST return an ID-* Fault Message and use `UnexpectedError` as the value of the `code` XML attribute.

A service specification MAY define more cases in which ID-* Fault Message is used.

Even if the processing of some parts of a message body fails, a WSP SHOULD always try to process the message body as well as it can according the specified processing rules and return normal response message indicating the failed parts in returned status codes (see Section 3.2.2) as one message may contain multiple task requests and succeeding in individual tasks is valuable, unless processing rules specify that after the first failed part the whole message should fail.

One *RequestElement* may contain number of individual task request (e.g., inside a **<Query>** there can be multiple **<QueryItem>** elements). So, after failing to complete one requested task, there could be a number of other tasks requested in the same message and a WSP SHOULD try to complete those unless service specific processing rules specify otherwise.

3.2.1. Top Level <Status> Element

A *ResponseElement* element contains one top level **<Status>** element to indicate whether or not the processing of a *RequestElement* has succeeded. The **<Status>** element is included from the Liberty Utility Schema. A **<Status>** element MAY contain other **<Status>** elements, providing more detailed information. A **<Status>** element has a `code` XML attribute, which contains the return status as a string. The local definition of these codes is specified in this document.

The `code` XML attribute of the top level **<Status>** element MUST contain one of the following values `OK`, `Partial` or `Failed`.

`OK` The value `OK` means that the processing of a *RequestElement* has succeeded. A second level status code MAY be used to indicate some special cases, but the processing of a *RequestElement* has succeeded.

`Partial` The value `Partial` means that the processing has succeeded only partially and partially failed, e.g., in the processing of a <**Query**> element some <**QueryItem**> element has been processed successfully, but the processing of some other <**QueryItem**> elements has failed. When the value `Partial` is used for the `code` XML attribute of the top level <**Status**> element, the top level <**Status**> element MUST have second level <**Status**> elements to indicate the failed parts of a *RequestElement*. The processing of the parts not referred to by any of the second level <**Status**> elements MUST have succeeded. A WSP MUST NOT use the value `Partial`, if it has not processed the whole *RequestElement*.

A WSP MUST NOT use the value `Partial` in case of modification requests, when a failed <**ModifyItem**> element didn't have a valid `itemID` XML attribute, i.e., a WSP is not able to indicate the failed <**ModifyItem**> element. In those cases a WSP MUST use the value `Failed` and anything changed based on the already processed part MUST be rolled back.

A WSP MAY also choose to fail completely another type of *RequestElement*, when only a part of it has failed, if the failed part does not have a valid `itemID` XML attribute. When ever the top level value `Failed` is used instead of `Partial` due to one or more missing `itemID` XML attributes, the second level status code `MissingItemID` MUST be used in addition to any other second level status code.

In some cases the most descriptive second level status code may not be used as it, for example, might compromise the privacy of a Principal. In those cases, when the second level status code must be used to indicate the failed parts in a case of a partial failure, the value `UnspecifiedError` MUST be used for the second level status code.

`Failed` The value `Failed` means that the processing of a *RequestElement* has failed. Either the processing of the whole *RequestElement* has totally failed or it might have succeeded partially, but the WSP decided to fail it completely. A specification for a service MAY also deny the use of the partial failure and so force a WSP to fail, even when it could partially succeed. A second level status code SHOULD be used to indicate the reason for the failure.

3.2.2. Second Level <Status> Codes

This specification defines the following second level status codes to be used as values for the `code` XML attribute:

```
ActionNotAuthorized
AggregationNotSupported
AllReturned
ChangeHistoryNotSupported
ChangedSinceReturnsAll
DataTooLong
DoesNotExist
EmptyRequest
ExistsAlready
ExtensionNotSupported
Failed
FormatNotSupported
InvalidData
InvalidExpires
InvalidItemIDRef
InvalidObjectType
InvalidPredefined
InvalidSelect
InvalidSetID
InvalidSetReq
InvalidSort
ItemIDDuplicated
```

```
  ResultQueryNotSupported
  MissingCredentials
  MissingDataElement
  MissingExpiration
  MissingItemID
  MissingNewDataElement
  MissingObjectType
  MissingSecurityMechIDElement
  MissingSelect
  ModifiedSince
  NewOrExisting
  NoMoreElements
  NoMoreObjects
  NoMultipleAllowed
  NoMultipleResources
  NoSuchTest
  ObjectTypeMismatch
  OK
  PaginationNotSupported
  Partial
  RequestedAggregationNotSupported
  RequestedPaginationNotSupported
  RequestedSortingNotSupported
  RequestedTriggerNotSupported
  SecurityMechIDNotAccepted
  SetOrNewQuery
  SortNotSupported
  StaticNotSupported
  TimeOut
  TriggerNotSupported
  UnexpectedError
  UnspecifiedError
  UnsupportedObjectType
  UnsupportedPredefined
```

If a request or notification fails for some reason, the `ref` XML attribute of the **<Status>** element SHOULD contain the value of the `itemID` XML attribute of the offending element in the request message. When the offending element does not have the `itemID` XML attribute, the reference SHOULD be made using the value of the `id` XML attribute, if that is present.

If it is not possible to refer to the offending element (as it has no `id`, or `itemID` XML attribute) the reference SHOULD be made to the ancestor element having a proper identifier XML attribute closest to the offending element.

When a WSC compose a request message, it SHOULD avoid using same value for any two XML attributes, which can be used to refer to the right place in return status. If there anyway are two XML attributes with the same value and a WSP needs to refer using either of them when indicating a problem, a WSP MAY consider the whole message as failed or used that value, when a high priority XML attribute has it. The priority order is `itemID`, `id`, so, for example, if both an `itemID` and an `id` has same value, it can be used to refer to the element having the `itemID` XML attribute with that value.

3.3. The `timeStamp` XML Attribute

A response and a notification message can also have a time stamp. This time stamp is provided so that the receiving party can later check whether there have been any changes since a response or a notification was received, or make modifications, which will only succeed if there have been no other modifications made after the time stamp was received.

The processing rule for the `timeStamp` XML Attribute

A WSP MUST add a `timeStamp` to a *ResponseElement*, if the processing of the *RequestElement* was successful and a WSP supports either the `changedSince` XML attribute or the `notChangedSince` XML attribute or both properly. The `timeStamp` XML attribute MUST have a value which can also be used as a value for the `changedSince` XML attribute, when querying changes made after the request for which the `timeStamp` was returned or the notification, which had the `timeStamp`. The value of the `timeStamp` XML attribute MUST also be such that it can be used as a value for the `notChangedSince` XML attribute, when making modifications after the request for which the `timeStamp` was returned or after receiving the notification message, which carried the `timeStamp` and the modifications will not succeed, if there has been any modification after this request or notification.

3.4. General Error Handling

A WSP MAY also register a relevant discovery option keyword to indicate that it does not support certain type of requests although they are available based on the specification for the service a WSP is hosting. Following discovery option keywords are specified for this purpose:

- urn:liberty:dst:noQuery
- urn:liberty:dst:noCreate
- urn:liberty:dst:noDelete
- urn:liberty:dst:noModify

A WSP may encounter problems other than errors in the incoming message:

1. If the processing takes too long (for example some back-end system is not responding fast enough) the second level status code `TimeOut` SHOULD be used to indicate this, when the request is not processed due to a WSP internal time out. The duration and indeed criteria for deciding when timeout has happened depend on WSP and are not externally visible other than the fact that the `TimeOut` status code is returned. Note that [LibertySOAPBinding] specifies a header block which a WSC may use to define threshold for timeout, but that is different functionality and the processing rules for that are specified in [LibertySOAPBinding].

2. Other error conditions than those listed in this specification and in service specifications may occur. There are two status codes defined for those cases. For cases a WSP (or WSC receiving a notification) can handle normally but for which there is no status code specified, the second level status code `UnspecifiedError` SHOULD be used. For totally unexpected cases the second level status code `UnexpectedError` SHOULD be used.

3.5. Linking with `ids`

Different types of `id` XML attributes are used to link queries and responses and notifications and acknowledgments together (see Figure 5). Response messages are correlated with requests using **<wsa:messageID>** and **<wsa:RelatesTo>** SOAP headers (see [LibertySOAPBinding]). Inside messages `itemID` and `itemIDRef` XML attributes are used for linking information inside response and acknowledgment messages to the details of request and notification messages.

See the definitions and the processing rules of the protocol elements for more detailed information.

Some elements in all messages can have `id` XML attributes of type `xs:ID`. These `id` XML attributes are necessary when some part of the message points to those elements. As an example, if usage directives are used, then the usage directive element must point to the correct element (see [LibertySOAPBinding]). Some parts of the messages may be signed and the `id` XML attribute is necessary to indicate which elements are covered by a signature.

It often happens that a query item of some sort needs to be correlated with a data item. The `itemID` and `itemIDRef` XML attributes are used for this purpose. They differ from regular XML ID attributes in that the namespace, and consequently the uniqueness constraint, are per type of item referred, i.e., same `itemID` can appear in **<TestItem>** and **<QueryItem>** without danger of confusion.

3.6. Resources

The present version of DST differs from previous versions, see Section 8, significantly in the way the resource is accessed: there is no explicit ResourceID anymore. The resource is identified by one of the following mechanisms

- Implicitly (e.g., PAOS exchange)
- From **<TargetIdentity>** SOAP header, see [LibertySOAPBinding]
- Using credentials that were supplied: it is presumed that the resource of the credential holder, i.e., the principal herself, is to be accessed.
- From endpoint. A service may choose to offer different end point for every resource accessed. The simplest case of this is to represent the resource as a part of the query string.

If confidentiality of the resource being accessed is desired, the **<TargetIdentity>** or the credentials, a SAML assertion inside **<wss:Security>** header, SHOULD contain an encrypted SAML assertion (this mechanism replaces the **<EncryptedResourceID>** mechanism of DST 1.1).

3.7. Selection

The second level of the selection is deeper inside the *RequestElement* element. The request message must describe in more detail what it wants to access inside the specified resource. This can be specified in two different ways. Either the requesting WSC accesses data by selecting it explicitly in the request or uses *predefined* selection. When the predefined selections are supported, the available predefined selections are specified in the service specification or are agreed out of band. A WSC specifies the predefined selection it wants to use by putting its identifier into the request. The identifier is carried as the value of the `predefined` XML attribute. When a WSC explicitly selects the data, it has to first specify the type of the data object it wants to access and then select the right objects and the data inside it. The XML attribute `objectType` and the element **<Select>** are specified for making the explicit selection.

```
<xs:element name="ChangeFormat">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="ChangedElements"/>
      <xs:enumeration value="CurrentElements"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
<xs:attribute name="changeFormat">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="ChangedElements"/>
      <xs:enumeration value="CurrentElements"/>
      <xs:enumeration value="All"/>
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>
<xs:attribute name="objectType" type="xs:NCName"/>
<xs:attribute name="predefined" type="xs:string"/>
<xs:attributeGroup name="selectQualif">
  <xs:attribute ref="dst:objectType" use="optional"/>
  <xs:attribute ref="dst:predefined" use="optional"/>
</xs:attributeGroup>
```

Figure 6. XML Attributes for <Select>

The name of the root element of an object is used as the identifier of that object type (XML attribute `objectType`). Each service specification must list the supported object types and provide their names, schemata and semantics. All object types starting by underscore character ("_") are reserved for use by Liberty framework specifications. Other than that, the namespace of object types is up to the service specification. When a service type supports only one

type of object, the `objectType` XML attribute may be left out from request messages. Also a service may specify a default object type, which is assumed, if the `objectType` XML attribute is not present.

As an example, when the resource is a personal profile, the **<Select>** can point to a home address. In the case of a **<Query>**, this means that the whole home address is requested, or for a **<Modify>**, the whole home address is being modified, etc. When only a part of a home address is accessed, the **<Select>** element must point only to that part, or in the case of a **<Modify>** the parts not to be modified must be rewritten using their existing values, when whole home address is given. Different parts of the resource can be accessed using the same *RequestElement* element as those elements can contain multiple **<Select>** elements in their own sub-structure.

Please note that the previous paragraph only described an example. The **<Select>** element may also be used differently. It is defined to contain needed parameters, but the parameters are defined by the specification for a service type. A service may have multiple different type of parameters characterizing data to be accessed and, for example, instead of pointing to some point in a data structure, the content of the **<Select>** element may, for example, list the data items to be accessed with some quality requirements for the data to be returned.

The **<Select>** element may also be omitted from a request, when all objects of the specified or default type are accessed, e.g., queried, in one request.

The type of **<Select>** is `SelectType`. Although the type is referenced by *this* specification, the type may vary according to the service specifications using this schema, and therefore MUST be defined within each service schema. As the type of the **<Select>** element may be quite different in different services, a service specification MUST specify the needed processing rules, if the processing rules provided by this specification are not adequate. If there are any conflicts the processing rules in the service specifications MUST override the processing rules in this specification.

When the `SelectType` is specified for a service, it must be very careful about what type of queries and modifies needs to be supported. Typically the **<Select>** points to some place in the conceptual XML document and it is RECOMMENDED that a string containing an XPath expression is used for **<Select>** element in those kind of cases. There are many other type of cases and the `SelectType` must be properly specified to cover the needs of a service type.

As a service may support different type of objects, the `SelectType` MUST be defined so that it supports all different types of objects.

When XPath is used, it is not always necessary to support full XPath. Services SHOULD limit the required set of XPath expressions in their specifications when full XPath is not required. A service may support full XPath even if it is not required. In that case the service MAY register the urn:liberty:dst:fullXPath discovery option keyword. If the required set of XPath expressions does not include the path to each element, a service may still support all paths without supporting full XPath. In that case the service MAY register the urn:liberty:dst:allPaths discovery option keyword.

3.8. Common Processing Rules for Selection

3.8.1. Processing Rules for the `predefined` XML Attribute

1. When a WSC uses the `predefined` XML attribute in a subelement of a *RequestElement* element, it MUST NOT use the `objectType` XML attribute, the **<Select>** element, or the **<Sort>** element. If either or all of them are present anyway together with a `predefined` XML attribute, a WSP MUST ignore them, when processing that subelement.

2. If the `predefined` XML attribute contains an identifier of a predefined selection, which a WSP does not support, the processing of the subelement containing the `predefined` XML attribute MUST fail and a status code indicating the failure MUST be returned in the response. A more detailed status code with the value `UnsupportedPredefined` SHOULD be used in addition to the top level status code. If the `predefined` XML attribute contains an unknown value, the processing of the subelement containing the `predefined` XML attribute MUST fail and a status code indicating failure MUST be returned in the response. A more detailed status code with the value `InvalidPredefined` SHOULD be used in addition to the top level status code.

3. A WSP MUST follow service specific processing rules for the values of the `predefined` XML attribute.

3.8.2. Processing Rules for the `objectType` XML Attribute

1. If the `objectType` XML attribute is missing from a subelement of a *RequestElement* element, when it is supposed to be used, the processing of that subelement MUST fail and a status code indicating the failure MUST be returned in the response. A more detailed status code with the value `MissingObjectType` SHOULD be used in addition to the top level status code. The subelements referred here are the **<QueryItem>**, the **<CreateItem>**, the **<DeleteItem>**, the **<ModifyItem>**, and the **<ResultQuery>**. All these elements are defined later with other protocol elements. Note: in some cases the `objectType` XML attribute is not needed, e.g., when a default object type has been defined for a service and that object type is accessed or a service only supports one `objectType`.

2. If the `objectType` XML attribute refers to a specified object type, but the WSP does not support it, the processing of the subelement containing the `objectType` XML attribute MUST fail. A more detailed status code with the value `UnsupportedObjectType` SHOULD be used in addition to the top level status code. If the `objectType` XML attribute contains an unknown object type name, the processing of the subelement containing the `objectType` XML attribute MUST fail. A more detailed status code with the value `InvalidObjectType` SHOULD be used in addition to the top level status code. Note that a data service may support extensions, making it difficult for a requestor to know the exact set of allowable values for the `objectType` XML attribute.

3.8.3. Processing Rules for the <Select> Element

1. If the **<Select>** element is missing from a subelement of a *RequestElement* element, when it is supposed to be use, the processing of that subelement MUST fail and a status code indicating the failure MUST be returned in the response. A more detailed status code with the value `MissingSelect` SHOULD be used in addition to the top level status code. The subelements referred here are the **<DeleteItem>**, the **<QueryItem>**, the **<ResultQuery>**, and the **<ModifyItem>**. All these elements are defined later with other protocol elements. Note: in some cases the **<Select>** element is not needed.

2. If the **<Select>** element has invalid content, e.g., does not match with the object type specified by the `objectType` XML attribute, contains an invalid pointer to a data not supported by the WSP or doesn't contain the specified parameters, the processing of the subelement containing the **<Select>** element MUST fail and a status code indicating failure MUST be returned in the response. A more detailed status code with the value `InvalidSelect` SHOULD be used in addition to the top level status code, unless a service specification specifies more detailed status codes better suited for the case. Note that a data service may support extensions, making it difficult for a requestor to know the exact set of allowable values for the **<Select>** element.

3.9. Requesting Meta and Additional Data

`ResultQueryType` and `ItemDataType` have an important role as parent classes of `QueryType` and **<Data>**, respectively.

When a WSC sends a request to create or modify data, it might want to get back some additional data in addition to the normal processing status, e.g., to get metadata a WSP has added to the newly created data. **<Create>** and **<Modify>** elements allow inclusion of **<ResultQuery>** elements in a request. A **<ResultQuery>** element is the basic data selection element and can contain normal selection parameters: XML attributes `predefined` and `objectType` and **<Select>** element. It may have also other parameters used in normal queries. These parameters and their processing

rules are introduced in Section 4. The data queried with one **<ResultQuery>** element is returned in one **<ItemData>** element.

<ItemData> is very similar to the **<Data>** element used to return data in responses to normal queries. The only difference is that the **<Data>** element can have more XML attributes as normal queries have more features like pagination. For the XML attributes common to both alternatives the same description and processing rules are valid, see Section 4 for details.

```
<xs:complexType name="ResultQueryBaseType">
  <xs:sequence>
    <xs:element ref="dst:ChangeFormat" minOccurs="0" maxOccurs="2"/>
  </xs:sequence>
  <xs:attributeGroup ref="dst:selectQualif"/>
  <xs:attribute ref="lu:itemIDRef" use="optional"/>
  <xs:attribute name="contingency" use="optional" type="xs:boolean"/>
  <xs:attribute name="includeCommonAttributes" use="optional" type="xs:boolean" default="0"/>
  <xs:attribute name="changedSince" use="optional" type="xs:dateTime"/>
  <xs:attribute ref="lu:itemID" use="optional"/>
</xs:complexType>
<xs:attributeGroup name="ItemDataAttributeGroup">
  <xs:attribute ref="lu:itemIDRef" use="optional"/>
  <xs:attribute name="notSorted" use="optional">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="Now"/>
        <xs:enumeration value="Never"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
  <xs:attribute ref="dst:changeFormat" use="optional"/>
</xs:attributeGroup>
```

Figure 7. XML Attributes and Base Type for ResultQuery and ItemData

```
<xs:element name="Select" type="dstref:SelectType"/>
<xs:element name="ResultQuery" type="dstref:ResultQueryType"/>
<xs:complexType name="ResultQueryType">
  <xs:complexContent>
    <xs:extension base="dst:ResultQueryBaseType">
      <xs:sequence>
        <xs:element ref="dstref:Select" minOccurs="0" maxOccurs="1"/>
        <xs:element name="Sort" minOccurs="0" maxOccurs="1" type="dstref:SortType"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:element name="ItemData" type="dstref:ItemDataType"/>
<xs:complexType name="ItemDataType">
  <xs:complexContent>
    <xs:extension base="dstref:AppDataType">
      <xs:attributeGroup ref="dst:ItemDataAttributeGroup"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Figure 8. Reference Model ResultQuery and ItemData

It is recommended that service specification writers study carefully when allowing requesting additional data provides enough benefits compared to separate queries to justify the additional complexity.

3.10. Common Processing Rules for Requesting Meta and Additional Data

1. A **<ResultQuery>** element MUST be processed as if it was a **<QueryItem>** element and the **<Data>** element used to carry the responses is replaced with **<ItemData>** taking into account the facts that failing **<ResultQuery>** elements do not usually cause a failure of the request message and that **<ResultQuery>** and **<ItemData>** have less features. See Section 4 for details.

2. If the processing of an **<ResultQuery>** element fails, the rest of the request message MUST be processed normally unless otherwise specified in the service specification. Proper second level status codes SHOULD be used indicate The reason for failing to process the **<ResultQuery>** element, but this MUST NOT affect the value of the top level status code unless otherwise specified in the service specification.

3. If a WSP does not support **<ResultQuery>** inside **<Create>** or **<Modify>** elements and it receives such, it MUST ignore it and process the message otherwise normally. Not responding to an **<ResultQuery>** is not considered failure and MUST NOT affect the value of the top level status code unless otherwise specified in the service specification. The second level status code `ResultQueryNotSupported` MUST be used to indicate that the WSP does not support this feature, if the feature is allowed in the service specification.

4. Each **<ResultQuery>** element MUST have the `itemID` XML attribute. Each **<ItemData>** element MUST have an `itemIDRef` XML attribute referring to the corresponding **<ResultQuery>** in the request.

5. A WSP MAY return additional data in a **<CreateResponse>** and a **<ModifyResponse>** without a WSC requesting for it. A WSC MUST tolerate such unsolicited **<ItemData>** even if it does not interpret it. Unsolicited **<ItemData>** MUST NOT have an `itemIDRef` XML attribute.

   Unsolicited data can be useful, if the WSP thinks that the WSC needs this data, e.g., to be able access the same data later on. For example a WSP may assign locally unique `id` to a newly created object and it wants to return it to the WSC so that the WSC could access the same object easily later on

6. If **<ResultQuery>** is used inside **<Create>** or **<Modify>** and it uses relative query expressions, the query MUST be interpreted relative to the data object just created or modified.

7. If **<ResultQuery>** is used inside **<Create>** or **<Modify>**, the `objectType` XML attribute of former MUST agree with the one in the latter.

4. Querying Data

Two different kinds of queries are supported; one for retrieving current data, and another for requesting only changed data. These two different kinds of queries can be present together in the same message. The response can contain the data with or without the common technical XML attributes, depending on the request. Some common XML attributes are always returned for some elements. When there are multiple elements matching the search criteria, they can be requested in smaller sets and sorted by defined criteria.

4.1. The <Query> Element

The **<Query>** element, which MAY appear multiple times in message body, unless forbidden by the service specification, has following sub-elements:

<TestItem> (optional) Test items, if present, can be used to specify tests over the data. A test evaluates to true or false without returning any data.

<QueryItem> (optional) Specifies what data the requestor wants from the resource and how. There can be multiple **<QueryItem>** elements in one **<Query>**. Or there could be none: in this case the query is evaluated only for purposes of the test items. A **<QueryItem>** can be *contingent* on a **<TestItem>** by referencing the latter via an ID. Often the data set used to evaluate the test will also be helpful for the query, e.g., the test can prime the cache for the query.

```
<xs:complexType name="TestItemBaseType">
  <xs:attributeGroup ref="dst:selectQualif"/>
  <xs:attribute name="id" use="optional" type="xs:ID"/>
  <xs:attribute ref="lu:itemID" use="optional"/>
</xs:complexType>
<xs:element name="TestResult" type="dst:TestResultType"/>
<xs:complexType name="TestResultType">
  <xs:simpleContent>
    <xs:extension base="xs:boolean">
      <xs:attribute ref="lu:itemIDRef" use="required"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

Figure 9. Utility Schema for TestItem and TestResult

```
<xs:complexType name="QueryType">
  <xs:complexContent>
    <xs:extension base="dst:RequestType">
      <xs:sequence>
        <xs:element ref="dstref:TestItem" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="dstref:QueryItem" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:element name="TestItem" type="dstref:TestItemType"/>
<xs:complexType name="TestItemType">
  <xs:complexContent>
    <xs:extension base="dst:TestItemBaseType">
      <xs:sequence>
        <xs:element name="TestOp" minOccurs="0" maxOccurs="1" type="dstref:TestOpType"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:element name="QueryItem" type="dstref:QueryItemType"/>
<xs:complexType name="QueryItemType">
  <xs:complexContent>
    <xs:extension base="dstref:ResultQueryType">
      <xs:attributeGroup ref="dst:PaginationAttributeGroup"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Figure 10. Reference Model for Query, TestItem, and QueryItem

4.1.1. The <TestItem> Element

The <**TestItem**> contains a <**TestOp**> qualified by some attributes. The two, in conjunction with `objectType` are used to indicate

1. the data on which the test is to be performed
2. the reference data against which the data (1) is to be tested
3. the nature of the test.

<TestOp> element

The content of the <**TestOp**>, the `TestOpType`, MUST be specified by the service specification that references DST.

For example, if service specification specifies XPath as query language and WSC wanted to ask whether or not the principal is of age, it could do so as follows:

```
<TestItem objectType="profile">
  <TestOp>//Age >= '21'</TestOp>
</TestItem>
```

In the above example, all 3 aspects of the test are expressed within the XPath expression that appears in <**TestOp**>.

Each <**TestItem**> evaluates to true or false depending on result of evaluation of the <**TestOp**>.

If service specification specifies XPath and <**TestOp**> does not indicate a top-level XPath boolean() function, the WSP MUST interpret the test expression as if this function was present.

Service Specific XPath Functions

Service specifications are encouraged to define XPath functions to simplify the expression of particular tests that are expected to be frequently requested. For instance, a profile specification might define a XPath function to simplify the of-age query:

```
number profile:age-compare([//age,] int test-age, string operator)
```

and permit selection like

```
<TestOp>profile:age-compare('21', 'gt')</TestOp>
```

Of course every service specific function requires service specific implementation, thus there is a continuum from XPath standard to slightly customized, to fully custom query languages and the service specification authors have to make the value judgment about where the sweet spot lies.

predefined XML attribute

While `objectClass` and **<TestOp>** aim to declaratively specify the test, in a specific deployment by mutual agreement of parties involved in message exchange, the `predefined` XML attribute can be used to specify some agreed test.

4.1.2. The <QueryItem> Element

The **<QueryItem>** element is a refinement of `ResultQueryType`, inheriting the `objectType` XML attribute and the **<Select>** and **<Sort>** elements as well as adding pagination related XML attributes.

The `objectType` and **<Select>** specify the data the query should return. The contents of the **<Select>** are determined by `SelectType` which MUST be defined by the service specification referencing DST.

When the **<Select>** defines that one or more data elements should be returned, then all of these elements (including their contained descendants) are returned unless service specific parameters filter out some or all requested data. Also privacy rules may not allow returning some or all of the requested data.

The **<QueryItem>** can also have a **<Sort>** element. The type and possible content of this element are specified by the services using this feature. The **<Sort>** element contains the criteria according to which the data in the response should be sorted. For example, address cards of a contact book could be sorted based on names using either ascending or descending order. As sorting is resource consuming the service specification MUST use sorting very carefully and specify sorting only based on the data and criteria which are really needed. In many cases sorting on the server side is not needed at all. When sorting is needed, only a very limited set of available sorting criteria should be defined.

The **<QueryItem>** can also have a **<ChangeFormat>** element (see Figure 6). The value of this element specifies, in which format the requesting WSC would like to have the data, when querying for changes. Two different formats are defined in this specification. These formats are explained in the processing rules (see Section 4.4).

The **<QueryItem>** element can have two XML attributes qualifying the query in more detail:

`includeCommonAttributes` (optional) The `includeCommonAttributes` specifies what kind of response is requested. The default value is `False`, which means that only the data specified in the service definition is returned. If the common XML attributes specified for container and leaf elements in this document are also needed, then this XML attribute must be given the value `True`. If the `id` XML attribute is used for distinguishing similar elements from one other by the service, it MUST always be returned, even if the `includeCommonAttributes` is `False`.

The `xml:lang` and `script` XML attributes are always returned when they exist.

`changedSince` (optional) The `changedSince` XML attribute should be used when the requestor wants to get only the data which has changed since the time specified by this XML attribute. The changed data can be returned in different ways. A WSC should specify the format it prefers using the element **<ChangeFormat>**. Please note that use of this `changedSince` XML attribute does not require a service to support the common XML attribute `modificationTime`. The service can keep track of the modification times without providing those times as `modificationTime` XML attributes for different data elements.

In addition to the `id` XML attribute, the **<ResultQuery>** or **<QueryItem>** element can also have an `itemID` XML attribute. The `itemID` XML attribute is correlated with `itemIDRef` XML attributes in the **<Data>** elements in the response to match the data to the **<QueryItem>** that produced them. Such correlation is necessary if the **<Query>** element contains multiple **<QueryItem>** elements.

4.1.3. Pagination

When the search criteria defined in the **<Select>** matches multiple elements of same type and name, the WSC may want to have the data in smaller sets, i.e., a smaller number of elements at a time. This is achieved by using the XML attributes `count`, `offset`, `setID` and `setReq` of the **<QueryItem>** element. The basic XML attributes are the `count` and the `offset`:

`count` (optional) The `count` XML attribute defines, how many elements should returned in a response. This is the amount of the elements directly addressed by the **<Select>**, their descendants are automatically included in the response, if not elsewhere otherwise specified.

`offset` (optional) The `offset` XML attribute specifies, from which element to continue, when querying for more data. The default value is zero, which refers to the first element.

```
<xs:attributeGroup name="PaginationAttributeGroup">
  <xs:attribute name="count" use="optional" type="xs:nonNegativeInteger"/>
  <xs:attribute name="offset" use="optional" type="xs:nonNegativeInteger" default="0"/>
  <xs:attribute name="setID" use="optional" type="lu:IDType"/>
  <xs:attribute name="setReq" use="optional">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="Static"/>
        <xs:enumeration value="DeleteSet"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
</xs:attributeGroup>
<xs:attributeGroup name="PaginationResponseAttributeGroup">
  <xs:attribute name="remaining" use="optional" type="xs:integer"/>
  <xs:attribute name="nextOffset" use="optional" type="xs:nonNegativeInteger" default="0"/>
  <xs:attribute name="setID" use="optional" type="lu:IDType"/>
</xs:attributeGroup>
```

Figure 11. XML Attributes for Pagination

Changes may happen while a WSC is requesting the data in smaller sets as this requires multiple <**Query**> messages and so will cause multiple <**QueryResponse**>s. This is not a problem for many services, but with some services this might cause problems as an inconsistent set of data may be returned to the requesting WSC. If supported by the service type and the WSP, a WSC may request that the modifications done by others are not allowed to effect what the requesting WSC gets. In the first <**Query**> of a sequence, the requesting WSC includes the `setReq` XML attribute with the value `Static`. The query response returns an identification for the set and in the following queries, this is included as the value of the `setID` XML attribute. At the end the WSC requests that the set is deleted (`setReq="DeleteSet"`) to free the resources on the WSP side.

`setID` (optional) The `setID` XML attribute contains an identification of a set. This must be used by a WSC, when it wants to make sure that no modifications are done to the set, while it is querying the data from the set.

`setReq` (optional) With the `setReq` XML attribute a WSC is able to request that a consistent set is created for coming queries (value `Static`) or a set is deleted (`DeleteSet`).

A service specification MUST specify the elements for which the pagination is supported. The pagination is not meant to be available for every request, just for a selected types of requests. As the use of the static sets may consume more resources on the server side than the normal pagination, the use of static sets must be considered carefully.

4.2. The <QueryResponse> Element

In addition to different identifiers the <**QueryResponse**> contains

<**Status**> Overall success or failure of the query

<**TestResult**> (optional) Indications of the outcomes of the test items that were present in the <**Query**>.

<**Data**> (optional) The data resulting from <**QueryItem**> elements. Each <**Data**> is correlated to corresponding <**QueryItem**> using `itemIDRef` XML attribute.

The <**QueryResponse**> elements are correlated, using their `itemIDRef` XML attributes, to the <**Query**> elements (`ItemID` XML attributes).

The requested data is encapsulated inside <**Data**> elements. One <**Data**> element contains data requested by one <**QueryItem**> element. If there were multiple <**QueryItem**> elements in the <**Query**>, the <**Data**> elements are linked to their corresponding <**QueryItem**> elements using the `itemIDRef` XML attributes.

If a WSC requested sorting, but a WSP does not support the requested type of sorting or sorting in general, a WSP SHOULD return the data unsorted, but then it MUST indicate this by including the XML attribute `notSorted` within the <**Data**> element carrying the unsorted data. The `notSorted` XML attribute may have either the value `Now`, when the requested sorting is not supported, but sorting in general is, or `Never`, when the sorting is not supported at all.

If a WSC was querying for changes, the <**Data**> element may contain the XML attribute `changeFormat` to indicate in which format the changes are returned (see Figure 6).

The <**Data**> element extends `ItemDataType` with XML attributes `nextOffset` and `remaining`, when a smaller set of the data instead all the data was requested using the `count` and the `offset` XML attributes in the request. The `nextOffset` XML attribute in a response is the offset of the first item not included in the response. So the value of the `nextOffset` XML attribute in a response can be used directly for the `offset` XML attribute in the next request, when the data is fetched sequentially. The `remaining` XML attribute defines, how many items there are after the last item included in the response. The `setID` XML attribute is also included, when a static set is accessed.

If there were multiple <**Query**> elements in the request message, the <**QueryResponse**> elements are linked to corresponding <**Query**> elements with `itemIDRef` XML attributes.

4.3. <ResultQuery> or <QueryItem> Conditioned by <TestItem>

ResultQueryType has itemIDRef and contingency attributes so that the query items can be made contingent on some <**TestItem**>. This itemIDRef correlates with the itemID in the <**TestItem**>, see Section 4.1.1

1. A service specification MAY restrict, or forbid, use of <**TestItem**> in conjunction with <**ResultQuery**> or <**QueryItem**>. if use of <**TestItem**> is fully supported, the WSP MAY register the discovery option keyword

   ```
   urn:liberty:dst:contingentQueryItems
   ```

2. If contingency attribute is present, then itemIDRef MUST be present as well and vice versa.

3. If the itemIDRef attribute does not match <**TestItem**> then the WSP MUST stop processing the <**QueryItem**> or <**ResultQuery**> and return a second level status code NoSuchTest.

4. If <**QueryItem**> or <**ResultQuery**> has a contingency attribute, the WSP MUST process the <**QueryItem**> or <**ResultQuery**> if and only if the <**TestItem**> referenced using the itemIDRef evaluates to the value of the contingency XML attribute.

5. The scope of the itemIDRef is one <**Query**>, <**Create**>, or <**Modify**>. itemIDRef MUST NOT refer to itemID in another top level element. The itemID XML attributes of <**TestItem**> elements MUST be unique within one <**Query**>, <**Create**>, or <**Modify**> element in the request. The <**TestItem**>, <**ResultQuery**>, and <**QueryItem**> share same itemID space.

4.4. Processing Rules for Queries

NOTE: The common processing rules specified earlier MUST also be followed (see Section 3).

4.4.1. Processing Rules for Multiple <QueryItem> Elements

One <**Query**> element can contain multiple <**QueryItem**> elements. The following rules specify how those must be supported and handled:

1. A WSP MUST support one <**QueryItem**> element inside a <**Query**> and SHOULD support multiple. If a WSP supports only one <**QueryItem**> element inside a <**Query**> and the <**Query**> contains multiple <**QueryItem**> elements, the processing of the whole <**Query**> MUST fail and a status code indicating failure MUST be returned in the response. A more detailed status code with the value NoMultipleAllowed SHOULD be used in addition to the top level status code. If a WSP supports multiple <**QueryItem**> elements inside a <**Query**>, it MAY register the urn:liberty:dst:multipleQueryItems discovery option keyword.

2. If the <**Query**> contains multiple <**QueryItem**> elements, the WSC MUST add itemID XML attributes to each <**QueryItem**> element. The WSP MUST link the <**Data**> elements to corresponding <**QueryItem**> elements using the itemIDRef XML attributes, if there were itemID XML attributes in the <**QueryItem**> elements and there were multiple <**QueryItem**> elements in the <**Query**>. The itemIDRef XML attribute in a <**Data**> element MUST have the same value as the itemID XML attribute in the corresponding <**QueryItem**> element.

3. If processing of a <**QueryItem**> fails, any remaining unprocessed <**QueryItem**> elements SHOULD NOT be processed. The data for the already processed <**QueryItem**> elements SHOULD be returned in the response message and the status code MUST indicate the failure to completely process the whole <**Query**>. A more detailed status SHOULD be used in addition to the top level status code to indicate the reason for failing to process the first failed <**QueryItem**>.

4. Unless service specification expressly allows an empty <**Query/**>, <**Query**> MUST have at least one <**QueryItem**> or <**TestItem**> element. If not, <**Query**> MUST fail with EmptyRequest second level code. If empty <**Query/**> is allowed, it SHOULD have semantics of returning the default document.

4.4.2. Processing Rules for <Select> Element

1. If there is no `changedSince` XML attribute in the **<QueryItem>** element and the **<Select>** requests valid data elements, but there are no values, the WSP MUST NOT return any **<Data>** element for that **<QueryItem>** unless a WSC is requesting pagination. In this case a WSP MUST return the **<Data>** element containing the `remaining` and the `nextOffset` XML attributes even, when no actual data is returned (see processing rules related to pagination later on).

2. If the **<Select>** requests multiple data elements, the WSP MUST return all of those data elements inside the **<Data>** element corresponding to the containing **<QueryItem>**.

4.4.3. Sorting Query Results

1. When the **<Sort>** element is included in a **<QueryItem>** element, the data returned inside a **<Data>** element SHOULD be sorted according to the criteria given in the **<Sort>** element. If a WSP doesn't support sorting, it SHOULD return the requested data unsorted. When the data is returned unsorted, the `notSorted` XML attribute MUST be used in the **<Data>** element containing the unsorted data. A WSP MAY also choose to fail to process the **<QueryItem>**, if it does not support sorting. In that case the second level status code `SortNotSupported` SHOULD be used in addition to the top level status code. A WSP may also register discovery option keyword urn:liberty:dst:noSorting, if the sorting has been specified for the service type, but the WSP doesn't support it.

2. If the content of the **<Sort>** element is not according to service specifications, a WSP SHOULD return the requested data unsorted. When the data is returned unsorted, the `notSorted` XML attribute MUST be used in the **<Data>** element containing the unsorted data and the second level status code `InvalidSort` SHOULD also be used. A WSP MAY also choose to fail to process the **<QueryItem>**, if the content of the **<Sort>** element is not according to service specifications. In this kind of a case the second level status code `InvalidSort` SHOULD be used in addition to the top level status code. If the content of the **<Sort>** element is valid, but a WSP does not support the requested type of sorting, it SHOULD return the requested data unsorted. When the data is returned unsorted, the `notSorted` XML attribute MUST be used in the **<Data>** element containing the unsorted data. A WSP MAY also choose to fail to process of the **<QueryItem>**, if it does not support the requested type of sorting. It SHOULD use the second level status code `RequestedSortingNotSupported` in addition to the top level status code.

3. When the `notSorted` XML attribute is used, it MUST have the value `Now`, when a WSP supports sorting, but not the requested type or the content of the **<Sort>** element was invalid. The `notSorted` XML attribute MUST have the value `Never`, when a WSP does not support sorting at all.

4.4.4. Pagination of Query Results

A WSC may want to receive the data in smaller sets instead of getting all the data at once, when there can be many elements with the same name. A WSC indicates this using either or both of the XML attributes `count` and `offset` in a **<QueryItem>** element, when the **<Select>** addresses a set of elements all having the same name. The number of elements inside this set may be restricted further by other parameters. Also access rights and policies may reduce the set of elements a WSC is allowed to get.

1. A WSP MUST always follow the same ordering, when the **<Select>** and **<Sort>** elements have the same values and either or both of XML attributes `count` and `offset` are used in the **<QueryItem>** element. If same query is made twice without a modification intervening, the result set MUST be the same and in same order. This is needed to make sure, for example, that a WSC really gets the next ten items, when asking for them, and not e.g. five of the previously returned items with five new items.

2. When either or both of the XML attributes `count` and `offset` is used in a **<QueryItem>** element and a WSP doesn't support pagination, the processing of whole **<QueryItem>** element MUST fail and the second level status code `PaginationNotSupported` SHOULD be used in addition to the top level status code. A WSP may support pagination, but not for the requested elements. In such a case the processing of whole **<QueryItem>** element MUST fail and the second level status code `RequestedPaginationNotSupported` SHOULD be used in addition to the top level status code. If a WSP doesn't support pagination at all, it MAY register the discovery option keyword urn:liberty:dst:noPagination to indicate this.

3. When the `count` XML attribute is included in a **<QueryItem>** element, the corresponding **<Data>** element in the **<QueryResponse>** MUST NOT contain more elements addressed with the value of the **<Select>** element than specified by the `count` XML attribute. A WSP MAY return a smaller number of elements of the same name that requested by a WSC. If the `count` XML attribute has the value zero, the WSP MUST NOT return any data elements inside the **<Data>** element. This `count="0"` may be used for querying the number of remaining elements starting from the specified offset, e.g., from offset zero, i.e., the total number of the elements addressed by the **<Select>** element. When the `count` XML attribute is not used in a **<QueryItem>** element, it means that the WSC requests for all data specified by other parameters like the **<Select>** element starting from the specified offset. As the default value for the `offset` XML attribute is zero, the case when neither of the XML attributes `offset` or `count` is not present reduces to a normal query.

4. When pagination is requested by a WSC, the elements inside a **<Data>** element MUST be in the ascending order of their offsets. The first element MUST have the offset specified by the `offset` XML attribute in the **<QueryItem>** element. The **<Data>** element MUST have both XML attributes `nextOffset` and `remaining`. The `nextOffset` XML attribute MUST have the offset of the first element not returned in the response. The value of the `remaining` XML attribute MUST define how many elements there are left starting from the value of the `nextOffset`, if a WSP knows that (e.g., that information might not be available from a backend system). If WSP does not know the exact value, it MUST use the value `-1` for the `remaining` XML attribute until it knows the value or there is no data left (`remaining="0"`). When `remaining="-1"`, a WSC must make new requests until `remaining="0"`, if it wants to get all the data.

5. Usually, when there is no data matching the different query parameters, no **<Data>** element is returned in a **<QueryResponse>**. When either or both of the `count` and `offset` attributes are used, the **<Data>** element MUST be returned, even, when no data is returned (e.g., no data available or `count="0"` used to get the number of data items). This is required so that a WSP can return the `remaining` and the `nextOffset` XML attributes to the requesting WSC.

6. When the `setReq` XML attribute is included in a **<QueryItem>** element and has the value `Static`, the WSP SHOULD return the `setID` XML attribute to the requesting WSC and process **<QueryItem>** elements later having this `setID` based on the data the WSP has at the time, when the value for the `setID` was created. If a WSP receives a **<QueryItem>** element having the `setReq` XML attribute and does not support static sets for the requested data or not at all, the processing of the **<QueryItem>** element MUST fail and a second level status code `StaticNotSupported` SHOULD be used in addition to the top level status code. If a WSP doesn't support static sets at all, it MAY register the discovery option keyword urn:liberty:dst:noStatic.

7. When the `setID` XML attribute is included in a request, the following parameters MUST NOT be used in a **<QueryItem>** element: the **<Select>** element, the **<Sort>** element, the `changedSince` XML attribute, the `includeCommonXML Attributes` XML attribute, or the `predefined` XML attribute. The requests are made from an earlier defined static set and the `count` and the `offset` XML attributes are used to define, what is requested from that set. If any of the mentioned parameters is present, when the `setID` XML attribute is used, it is unclear what a WSC wants and the processing of the whole **<QueryItem>** MUST fail and a second level status code `SetOrNewQuery` SHOULD be used in addition to the top level status code.

8. When the `setID` XML attribute is included in a **<QueryItem>** element and has a valid value, the **<Data>** element in the response MUST always have the `setID` XML attribute.

9. When a static set is created, the requesting WSC SHOULD query all the data it needs from this set as soon as possible and delete the static set immediately after this using `setReq="DeleteSet"`. A WSP MAY also delete the static set, even if a WSC hasn't yet requested the deletion of the static set. If a WSC tries to make a request to a non-existing static set, the processing of the whole <**QueryItem**> MUST fail and the second level status code `InvalidSetID` SHOULD be used in addition to the top level status code.

10. The `setReq="Static"` and the `setID` XML attribute MUST NOT be used simultaneously in a <**QueryItem**> element. If they are used, the WSP MUST ignore the `setReq="Static"` and process the <**QueryItem**> element like the `setReq` XML attribute would not be present.

11. If the `setReq` XML attribute has some other value than `Static` or `DeleteSet`, the processing of the whole <**QueryItem**> element must fail and a second level status code `InvalidSetReq` SHOULD be used in addition to the top level status code.

4.4.5. Effect of Access and Privacy Policies

Even when the requested data exists, it should be noted that access and privacy policies specified by the resource owner may cause the request to result in data not being returned to the requestor.

When a WSP processes a <**QueryItem**>, it MUST check whether the resource owner (the Principal, for example) has given consent to return the requested information. To be able to check WSC specific access rights, the WSP MUST authenticate the WSC (see [LibertySecMech]). The WSP MUST also check that any usage directive given in the request is acceptable based on the usage directives defined by the resource owner (see [LibertySOAPBinding]). If either check fails for any piece of the requested data, the WSP MUST NOT return that piece of data. Note that there can be consent for returning some data element, but not its XML attributes. For example, a resource owner might not want to release the `modifier` XML attribute, if she does not want to reveal information about which services she uses. The data for which there is no consent from the resource owner MUST be handled as if there was no data. The WSP MAY try to get consent from the resource owner while processing the request, e.g., by using an interaction service, see [LibertyInteract]. A WSP might check the access rights and policies in usage directives at a higher level, before getting to DST processing and MAY, in this case, just return an ID-* Fault Message [LibertySOAPBinding] without processing the <**Query**> element at all, if the requesting WSC is not allowed to access the data.

4.4.6. Querying Changes Since Specified Time

It is possible to query changes since a specified time using the `changedSince` XML attribute.

1. If the <**QueryItem**> element contains the `changedSince` XML attribute, the WSP SHOULD return only those elements addressed by the <**Select**> which have been modified since the time specified in the `changedSince` XML attribute. There are two different formats, in which the changed data can be returned. A WSC SHOULD indicate using the <**ChangeFormat**> element the format it prefers and also, if it understands the other format. The two formats are `ChangedElements` and `CurrentElements`. If a service specification doesn't specify anything else the value `ChangedElements` MUST be used as a default value as it is compatible with the format used in the version 1.0 of the Data Services Template.

2. A WSP MUST ignore the <**ChangeFormat**> element, if the `changedSince` XML attribute is not used in the same <**QueryItem**> element. A WSP MUST NOT use a format, which a WSC does not understand. Note that format `ChangedElements`, has the format `All` as a fallback solution, when a WSP doesn't have all the needed change history information. Also if a WSP doesn't support requesting only changed data, it returns all data.

3. A <**QueryItem**> element MAY contain two <**ChangeFormat**> element with different values. A WSP SHOULD use the format specified by the first <**ChangeFormat**> element, but, if it does not support that format, it MAY use the format specified by the second <**ChangeFormat**> element.

4. If a WSP does not support the format a WSC is requesting to be used, the processing of the **<QueryItem>** MUST fail and the second level status code `FormatNotSupported` SHOULD be used in addition to the top level status code.

5. If a WSC requests the `ChangedElements` format and a WSP supports it, the WSP SHOULD return only the changed information. If some element has been deleted, a WSP SHOULD return an empty element to indicate the deletion (**<ElementName+/>**). The only allowed exception to this is that the WSP does not have enough history information available to be able to return only the changed information. In that case it MUST use format `All` and return all current elements with their values even if those have not changed since the specified time.

6. If a WSC requests the `CurrentElements` format and a WSP supports it, the WSP SHOULD return only the currently existing elements. It SHOULD return an empty element, if the element has not changed, to indicate that no change has happened (**<ElementName/>**).

   N.B. As empty elements are used to indicate either deleted or not changed elements depending on the used format, the formats `CurrentElements` and `ChangedElements` do not work well, if the data hosted by a service may contain empty elements. In those cases a service should either use only format `All` or always have some XML attributes for the otherwise empty elements.

7. If a WSC has used the **<ChangeFormat>** element in a request, a WSP MUST use the `changeFormat` XML attribute in the response to indicate, which format is used. A WSP MUST not use the `changeFormat` XML attribute in a response, if the **<ChangeFormat>** element was not used in the corresponding request so the processing stays version 1.0 compatible, when the **<ChangeFormat>** element is not used.

8. If there can be multiple elements with same name, the `id` XML attribute or some other XML attribute used to distinguish the elements from each other MUST be included (e.g., in case of an ID-SIS Personal Profile service the following empty element could be returned **<AddressCard id="tr7632q"/>** to indicate a deleted or not changed **<AddressCard>** depending on the used format). If the value of the `id` XML attribute or some other XML attribute used for distinguishing elements with same name is changed, the WSP MUST consider this as a case, in which the element with the original value of the distinguishing XML attribute is deleted and a new one with the new value of the distinguishing XML attribute is created. To avoid this, a WSP MAY refuse to accept modifications of a distinguishing XML attribute and MAY require that an explicit deletion of the element is done and a new one created.

9. If the elements addressed by the **<Select>** have some values, but there has been no changes since the time specified in the `changedSince` XML attribute, the WSP MUST return empty **<Data>** element (**<Data/>**), when it returns the changes properly. This empty **<Data>** element indicates that no changes have occurred. There might be cases in which the WSP is not able to return changes properly, see later processing rules. Please note that in cases that have no values, no **<Data>** element is returned to indicate this. So empty **<Data>** element has different semantics than missing **<Data>** element.

10. If the **<QueryItem>** element contains the `changedSince` XML attribute and a WSP is not keeping track of modification times, it SHOULD process the **<QueryItem>** element as there would be no `changedSince` XML attribute, and indicate this in the response using the second level status code `ChangedSinceReturnsAll`. This is not considered a failure and the rest of the **<QueryItem>** elements MUST be processed. Also it might be that a WSP does not have a full change history and so for some queries, it is not possible to find out, which changes occurred after the specified time. As processing with access rights and policy in place might be quite complex, a WSP might sometimes process the query for changes properly and sometime process it as if there were no `changedSince` XML attribute. In those cases, when a WSP returns all current values, it SHOULD indicate this with the second level status code `AllReturned` and, if the **<ChangeFormat>** element was used in the request, the `changeFormat` XML attribute with the value `All` SHOULD be used. This is also not considered a failure and the rest of the **<QueryItem>** elements MUST be processed. Please note that the status code `AllReturned` differs from the status code `ChangedSinceReturnsAll`, as `ChangedSinceReturnsAll` means that the WSP never processes the `changedSince` XML attribute properly. A WSP MUST use either `AllReturned` or `ChangedSinceReturnsAll` as the second level status code, when it returns data, but does

not process the `changedSince` XML attribute properly, i.e., returns only the changes. If a WSP will not process the **<QueryItem>** elements with a `changedSince` XML attribute at all, it MUST indicate this with top level status code `Failed` and SHOULD also return a second level status code of `ChangeHistoryNotSupported` in the response. In this case a WSP MUST NOT return any **<Data>** element for the **<QueryItem>** element containing the `changedSince` XML attribute. If a WSP processes the `changedSince` XML attribute, it MUST also support the `notChangedSince` XML attribute for **<ModifyItem>** element and MAY register the urn:liberty:dst:changeHistorySupported discovery option keyword. Please note that still in some cases a WSP MAY return `AllReturned`.

11. Access rights and policies in place may affect how the queries for changes can work as they affect which elements and XML attributes a WSC is allowed to see. If a WSC was originally allowed to get the requested data, but is no longer after some change in access policies, then from its point of view that data is deleted and that should be taken into account in the response. If the WSP notices that access rights have changed, and the current rights do not allow access, it MUST return all data except the data for which the access rights were revoked, and use the second level status code `AllReturned` and, if the **<ChangeFormat>** element was used in the request, the `changeFormat` XML attribute with the value `All` SHOULD be used. The WSP MUST NOT return empty elements for the data for which access rights were changed even if the format `ChangedElement` was requested, as this might reveal the fact that this specific data has at least existed at the service in some point of time. Please note that it might be the case that the data was added after the WSCs access rights were revoked and the WSC was never supposed to be aware of the existence of that data. If the WSP notices that the access rights are changed and the current rights do allow access, it MUST consider the data for which the access rights are changed, as if it were just created.

12. Both the WSC and WSP may have policies specified by the Principal for control of their data. Only by comparing policy statements made by the WSC (via **<UsageDirective>** elements (see [LibertySOAPBinding]) with policies maintained on behalf of the Principal by the WSP it is possible to fully determine the effects of interaction between these sets of policies. As it might be too expensive to search for policies the WSC promised to honor when it made the original request, and this information might not even be available, the WSP might be only capable of making the decision based on the policy changes made by the Principal. If some data is prevented from being returned to the WSC due to conflicts in policies and the WSP notices that the Principal's policies have changed, it MUST return all data except that for which the Principal's policy has denied access against the current policy of a requesting WSC, and use the second level status code `AllReturned` to indicate that the WSC must check the response carefully to find out what has changed. Also if the **<ChangeFormat>** element was used in the request, the `changeFormat` XML attribute with the value `All` SHOULD be used. The WSP MUST NOT return empty elements for the data for which the Principal's policy was changed even if the format `ChangedElements` was requested, as this might reveal the fact that this specific data was exposed by the service at some point in time. Please note that it might be the case that that data has been added after the policies were changed and the requesting WSC was never supposed to be aware of that data, unless it changed the policy it promises to honor. If the WSP notices that the Principal's policy has changed and the current policy does allow access, it MUST consider the data for which the policy is changed as if it had been just created. If a WSC changes the policy it promises to honor, it SHOULD make a new query without a `changedSince` XML attribute.

13. As mentioned earlier, the WSP might in some cases return all the current data the **<Select>** points to, and not just the changes using specified format, even when the `changedSince` XML attribute is present. So the WSC MUST compare the returned data to previous data it had queried earlier to find out what really has changed. Note that this MUST be done even when the WSP has processed the `changedSince` correctly, because some values might have been changed back and forth and now they have same values that they used to have earlier, despite the most current previous values being different.

4.4.7. Requesting Common XML Attributes

The common XML attributes are not always returned. A WSC may indicate with the `includeCommonAttributes` XML attribute, whether it wants to have the common XML attributes or not.

1. If the `includeCommonAttributes` is set to `True`, the common XML attributes specified by XML attribute groups `commonAttributes` and `leafAttributes` MUST be included in the response, if their values are specified for the requested data elements. The `ACC` XML attributes MAY be left out, if the value is urn:liberty:dst:acc:unknown.

2. If the `id` XML attribute is used for distinguishing similar elements from each other by the service, it MUST be returned, even if the `includeCommonAttributes` is false. Also, when either or both of the XML attributes `xml:lang` and `script` are present, they MUST be returned, even if the `includeCommonAttributes` is false

4.5. Examples

The following query example, based on hypothetical profile service, requests the common name and home address of a Principal:

```
<hp:Query xmlns:hp="urn:liberty:hp:2005-07">
  <hp:QueryItem itemID="name">
    <hp:Select>/hp:HP/hp:CommonName</hp:Select>
  </hp:QueryItem>
  <hp:QueryItem itemID="home">
    <hp:Select>
      /hp:HP/hp:AddressCard
        [hp:AddressType="urn:liberty:id-sis-hp:addrType:home"]
    </hp:Select>
  </hp:QueryItem>
</hp:Query>
```

This query may generate the following response:

```
<hp:QueryResponse xmlns:hp="urn:liberty:hp:2005-07">
  <hp:Status code="OK"/>
  <hp:Data itemIDRef="name">
    <hp:CommonName>
      <hp:CN>Zita Lopes</hp:CN>
        <hp:AnalyzedName nameScheme="firstlast">
          <hp:FN>Zita</hp:FN>
          <hp:SN>Lopes</hp:SN>
          <hp:PersonalTitle>Dr.</hp:PersonalTitle>
        </hp:AnalyzedName>
        <hp:AltCN>Maria Lopes</hp:AltCN>
        <hp:AltCN>Zita Maria Lopes</hp:AltCN>
    </hp:CommonName>
  </hp:Data>
  <hp:Data itemIDRef="home">
    <hp:AddressCard id='9812'>
      <hp:AddressType>
        urn:liberty:id-sis-hp:addrType:home
      </hp:AddressType>
      <hp:Address>
        <hp:PostalAddress>
          c/o Carolyn Lewis$2378 Madrona Beach Way North
        </hp:PostalAddress>
        <hp:PostalCode>98503-2341</hp:PostalCode>
        <hp:L>Olympia</hp:L>
        <hp:ST>wa</hp:ST>
        <hp:C>us</hp:C>
      </hp:Address>
    </hp:AddressCard>
  </hp:Data>
</hp:QueryResponse>
```

If there was no user consent for the release of the **<hp:CommonName>** or for the whole **<hp:AddressCard>** with [hp:AddressType="urn:liberty:id-sis-hp:addrType:home"], apart from the country information, then the response is as follows (including a timestamp, as this service supports change history):

```
<hp:QueryResponse
   xmlns:hp="urn:liberty:hp:2005-07"
   timeStamp="2003-02-28T12:10:12Z">
  <hp:Status code="OK"/>
  <hp:Data itemIDRef="home">
    <hp:AddressCard id='9812'>
      <hp:AddressType>
        urn:liberty:id-sis-hp:addrType:home
      </hp:AddressType>
      <hp:Address><hp:C>us</hp:C></hp:Address>
    </hp:AddressCard>
  </hp:Data>
</hp:QueryResponse>
```

If there was no **<hp:CommonName>** and no **<hp:AddressCard>** with [hp:AddressType="urn:liberty:id-sis-hp:addrType:home"], then the response is:

```
<hp:QueryResponse
   xmlns:hp="urn:liberty:hp:2005-07"
   timeStamp="2003-02-28T12:10:12Z">
  <hp:Status code="OK"/>
</hp:QueryResponse>
```

The following request queries the fiscal identification number of the Principal with the common XML attributes:

```
<hp:Query xmlns:hp="urn:liberty:hp:2005-07">
  <hp:QueryItem includeCommonAttributes="True">
    <hp:Select>/hp:HP/hp:LegalIdentity/hp:VAT</hp:Select>
  </hp:QueryItem>
</hp:Query>
```

This query may generate the following response:

```
<hp:QueryResponse
   xmlns:hp="urn:liberty:hp:2005-07"
   id="12345"
   timeStamp="2003-05-28T23:10:12Z">
  <hp:Status code="OK"/>
  <hp:Data>
    <hp:VAT
       modifier="http://www.accountingservices.com"
       modificationTime="2003-04-25T15:42:11Z"
       ACC="urn:liberty:dst:acc:secondarydocuments">
      <hp:IDValue
       modifier="http://www.accountingservices.com"
       modificationTime="2003-04-25T15:42:11Z"
       ACC="urn:liberty:dst:acc:secondarydocuments">
         502677123
      </hp:IDValue>
      <hp:IDType
       modifier="http://www.accountingservices.com"
       modificationTime="2003-03-12T09:12:09Z"
       ACC="urn:liberty:dst:acc:secondarydocuments">
         urn:liberty:altIDType:itcif
      </hp:IDType>
    </hp:VAT>
```

```
  </hp:Data>
 </hp:QueryResponse>
 <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
  . . .
 </ds:Signature>
```

The following request queries for address information which has been changed since 12:10:12 28 February 2003 UTC:

```
 <hp:Query xmlns:hp="urn:liberty:hp:2005-07">
  <hp:QueryItem changedSince="2003-02-28T12:10:12Z">
   <hp:Select>/hp:HP/hp:AddressCard</hp:Select>
  </hp:QueryItem>
 </hp:Query>
```

This query can generate following response:

```
 <hp:QueryResponse
    xmlns:hp="urn:liberty:hp:2005-07"
    timeStamp="2003-05-30T16:10:12Z">
  <hp:Status code="OK"/>
  <hp:Data>
   <hp:AddressCard id='9812'>
    <hp:Address>
     <hp:PostalAddress>
      2891 Madrona Beach Way North
     </hp:PostalAddress>
    </hp:Address>
   </hp:AddressCard>
   <hp:AddressCard id='w1q2'/>
  </hp:Data>
 </hp:QueryResponse>
```

Please note that only the changed information inside the **<hp:AddressCard>** is returned. The response shows that after the specified time, there was also another **<hp:AddressCard>** present, but that has been deleted. As there can be many **<hp:AddressCard>** elements, the `id` XML attribute is returned to distinguish distinct elements.

If there have been no changes since the specified time, then the response is just:

```
 <hp:QueryResponse
    xmlns:hp="urn:liberty:hp:2005-07"
    timeStamp="2003-05-30T16:10:12Z">
  <hp:Status code="OK"/>
  <hp:Data/>
 </hp:QueryResponse>
```

If the same request for changed addresses is made including the **<hp:ChangeFormat>** element:

```
 <hp:Query xmlns:hp="urn:liberty:hp:2005-07">
  <hp:QueryItem changedSince="2003-02-28T12:10:12Z">
     <hp:Select>/hp:HP/hp:AddressCard</hp:Select>
     <hp:ChangeFormat>CurrentElements</hp:ChangeFormat>
  </hp:QueryItem>
 </hp:Query>
```

All the current elements are returned in the response:

```
<hp:QueryResponse
   xmlns:hp="urn:liberty:hp:2005-07"
   timeStamp="2003-05-30T16:10:12Z">
  <hp:Status code="OK"/>
  <hp:Data changeFormat="CurrentElements">
    <hp:AddressCard id='9812'>
      <hp:Address>
        <hp:PostalAddress>2891 Madrona Beach Way North</hp:PostalAddress>
        <hp:PostalCode/>
        <hp:L/>
        <hp:ST/>
        <hp:C/>
      </hp:Address>
    </hp:AddressCard>
  </hp:Data>
</hp:QueryResponse>
```

Please note that now all the current elements inside the **<hp:AddressCard>** are returned. The deleted **<hp:AddressCard>** is not shown at all and for the elements, which have not changed - only empty elements are returned.

If a WSP does not support change history, then the response could be:

```
<hp:QueryResponse
   xmlns:hp="urn:liberty:hp:2005-07"
   timeStamp="2003-05-30T16:10:12Z">
  <hp:Status code="OK">
    <Status code="ChangeSinceReturnsAll"/>
  </hp:Status>
  <hp:Data changeFormat="All">
    <hp:AddressCard id='9812'>
      <hp:AddressType>urn:liberty:id-sis-hp:addrType:home</hp:AddressType>
      <hp:Address>
        <hp:PostalAddress>2891 Madrona Beach Way North</hp:PostalAddress>
        <hp:PostalCode>98503-2341</hp:PostalCode>
        <hp:L>Olympia</hp:L>
        <hp:ST>wa</hp:ST>
        <hp:C>us</hp:C>
      </hp:Address>
    </hp:AddressCard>
  </hp:Data>
</hp:QueryResponse>
```

The rest of the examples are related to pagination and sorting based on fictional address service, so all the DST elements in the namespace of that fictional address service.

Parameters **<Select>** and **<Sort>** and returned **<Data>** elements do not have valid contents in the examples as the main point is to show the principle how pagination works and the use of the pagination related XML attributes

A Resource contains 40 address cards and the WSC A wants to list those ordered by the City and 10 at the time. Due to access rights and policies the WSC A is allowed to get only 30 of those AddressCards. The WSC A makes the first query:

```
<ads:Query xmlns:ads="http://www.example.com/2010/12/Addr">
  <ads:QueryItem count="10">
     <ads:Select>Pointing to the AddressCards</ads:Select>
     <ads:Sort>Requesting sorting by the City</ads:Sort>
  </ads:QueryItem>
</ads:Query>
```

and gets back the first ten address cards ordered by the City:

```
<ads:QueryResponse
   xmlns:ads="http://www.example.com/2010/12/Addr"
   timeStamp="2004-03-23T03:40:00Z">
  <ads:Status code="OK"/>
  <ads:Data remaining="20" nextOffset="10">first ten address cards</ads:Data>
</ads:QueryResponse>
```

Then it queries the next ten starting from offset 10:

```
<ads:Query xmlns:ads="http://www.example.com/2010/12/Addr">
  <ads:QueryItem count="10" offset="10">
     <ads:Select>Pointing to the AddressCards</ads:Select>
     <ads:Sort>Requesting sorting by the City</ads:Sort>
  </ads:QueryItem>
</ads:Query>
```

and gets those

```
<ads:QueryResponse
   xmlns:ads="http://www.example.com/2010/12/Addr"
   timeStamp="2004-03-23T03:40:20Z">
  <ads:Status code="OK"/>
  <ads:Data remaining="10" nextOffset="20">next ten address cards</ads:Data>
</ads:QueryResponse>
```

After this the WSC B adds one more address card to the resource. The WSC A is allowed to get this address card, but when sorting based on the City, this new card has the offset 15. When the WSC A fetches the next ten address cards:

```
<ads:Query xmlns:ads="http://www.example.com/2010/12/Addr">
  <ads:QueryItem count="10" offset="20">
     <ads:Select>Pointing to the AddressCards</ads:Select>
     <ads:Sort>Requesting sorting by the City</ads:Sort>
  </ads:QueryItem>
</ads:Query>
```

It gets ten address cards, but it has already received the first address card already in the previous response.

```
<ads:QueryResponse
   xmlns:ads="http://www.example.com/2010/12/Addr"
   timeStamp="2004-03-23T03:41:00Z">
  <ads:Status code="OK"/>
  <ads:Data remaining="1" nextOffset="30">next ten address cards</ads:Data>
</ads:QueryResponse>
```

Finally the WSC A fetches the last address card.

```
<ads:Query xmlns:ads="http://www.example.com/2010/12/Addr">
  <ads:QueryItem count="1" offset="30">
     <ads:Select>Pointing to the AddressCards</ads:Select>
     <ads:Sort>Requesting sorting by the City</ads:Sort>
```

```
    </ads:QueryItem>
  </ads:Query>
```

and gets the 31st address card from offset 30.

```
  <ads:QueryResponse
     xmlns:ads="http://www.example.com/2010/12/Addr"
     timeStamp="2004-03-23T03:41:17Z">
    <ads:Status code="OK"/>
    <ads:Data remaining="0" nextOffset="31">the last address card</ads:Data>
  </ads:QueryResponse>
```

So the WSC A didn't get this new address card added by the WSC B and got one card twice.

In an alternative scenario, if supported by the WSP, the WSC A requests a static set to be created so that simultaneous modifications can not affect the results the WSC A gets. The initial request includes the `setReq` XML attribute:

```
  <ads:Query xmlns:ads="http://www.example.com/2010/12/Addr">
    <ads:QueryItem count="10" setReq="Static">
       <ads:Select>Pointing to the AddressCards</ads:Select>
       <ads:Sort>Requesting sorting by the City</ads:Sort>
    </ads:QueryItem>
  </ads:Query>
```

In the response the first ten address cards are returned together with a handle to this static set (the XML attribute `setID`).

```
  <ads:QueryResponse
     xmlns:ads="http://www.example.com/2010/12/Addr"
     timeStamp="2004-03-23T03:40:00Z">
    <ads:Status code="OK"/>
    <ads:Data remaining="20" nextOffset="10" setID="gfkjds98">
      first ten address cards
    </ads:Data>
  </ads:QueryResponse>
```

In the next query the WSC A queries the next ten address card referring to the static set using the `setID` XML attribute. The **<Select>** element is not anymore used.

```
  <ads:Query xmlns:ads="http://www.example.com/2010/12/Addr">
    <ads:QueryItem count="10" offset="10" setID="gfkjds98"/>
  </ads:Query>
```

In the response the next ten address cards are returned and the `setID` is still returned as always when accessing a static set.

```
  <ads:QueryResponse
     xmlns:ads="http://www.example.com/2010/12/Addr"
     timeStamp="2004-03-23T03:40:00Z">
    <ads:Status code="OK"/>
    <ads:Data remaining="10" nextOffset="20" setID="gfkjds98">
      next ten address cards
    </ads:Data>
  </ads:QueryResponse>
```

When the WSC B tries to add a new address card, it doesn't affect the data the WSC A gets, when requesting the next ten address cards.

```
<ads:Query xmlns:ads="http://www.example.com/2010/12/Addr">
  <ads:QueryItem count="10" offset="20" setID="gfkjds98"/>
</ads:Query>
```

So the WSC A gets the last ten address card.

```
<ads:QueryResponse
   xmlns:ads="http://www.example.com/2010/12/Addr"
   timeStamp="2004-03-23T03:40:00Z">
  <ads:Status code="OK"/>
  <ads:Data remaining="0" nextOffset="30" setID="gfkjds98">
    ... next ten address cards ...
  </ads:Data>
</ads:QueryResponse>
```

Finally the WSC A deletes the static set. This deletion could have been done together with the previous request, but the WSC wanted to play safe and delete the static set only after getting all the data it wanted.

```
<ads:Query xmlns:ads="http://www.example.com/2010/12/Addr">
  <ads:QueryItem count="0" setID="gfkjds98" setReq="DeleteSet"/>
</ads:Query>
```

And the WSP acknowledges the request.

```
<ads:QueryResponse
   xmlns:ads="http://www.example.com/2010/12/Addr"
   timeStamp="2004-03-23T03:40:00Z">
  <ads:Status code="OK"/>
</ads:QueryResponse>
```

So the addition the WSC B tried to make is not visible in the static set. Either the WSP refused to accept the addition while WSC A was accessing the data or it created a temporary set for the WSC A to access and the modification by the WSC B was accepted, but not visible in the temporary static set created for WSC A. In the example above the WSP created a temporary set an so returned the same time stamp in all responses containing data from that temporary set.

5. Creating Data Objects

A WSC can create new data objects to a resource when a service type supports multiple objects of the same type. If there is only one object of a type, that object exists always, when a resource containing it exists. The data objects can later be modified and deleted.

5.1. <Create> Element

The **<Create>** element is used to create new data objects, not new data inside existing data objects. The content of a data object is created, deleted and modified using the **<Modify>**. The right resource, to which a new data object is added, is selected using security mechanism and possibly **<TargetIdentity>** header. The **<CreateItem>** element specifies the type of the new object (the `objectType` XML attribute) and initial content for the new object (inside the **<NewData>** element). The **<NewData>** MAY contain some local addressing element that further qualifies the object that is being created. For example, when adding an address card, service specification may specify an address card identifier that differentiates the object from other similar objects (or this identifier may be assigned automatically by the service, in which case the **<ResultQuery>** may come handy to discover which identifier was assigned).

```
<xs:attributeGroup name="CreateItemAttributeGroup">
  <xs:attribute ref="dst:objectType" use="optional"/>
  <xs:attribute name="id" use="optional" type="xs:ID"/>
  <xs:attribute ref="lu:itemID" use="optional"/>
</xs:attributeGroup>
```

Figure 12. XML Attributes for CreateItem

```
<xs:complexType name="CreateType">
  <xs:complexContent>
    <xs:extension base="dst:RequestType">
      <xs:sequence>
        <xs:element ref="dstref:CreateItem" minOccurs="1" maxOccurs="unbounded"/>
        <xs:element ref="dstref:ResultQuery" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:element name="CreateItem" type="dstref:CreateItemType"/>
<xs:complexType name="CreateItemType">
  <xs:sequence>
    <xs:element ref="dstref:NewData" minOccurs="0" maxOccurs="1"/>
  </xs:sequence>
  <xs:attributeGroup ref="dst:CreateItemAttributeGroup"/>
</xs:complexType>
<xs:element name="NewData" type="dstref:AppDataType"/>
<xs:complexType name="CreateResponseType">
  <xs:complexContent>
    <xs:extension base="dstref:DataResponseType"/>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="DataResponseType">
  <xs:complexContent>
    <xs:extension base="dst:DataResponseBaseType">
      <xs:sequence>
        <xs:element ref="dstref:ItemData" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Figure 13. Reference Model for Create

5.2. <CreateResponse> Element

The <**CreateResponse**> element contains in addition to the mandatory <**Status**> element possible <**ItemData**> elements, which carry requested data related to the data just created. For example, returned data could include a unique ID assigned to the data object just created.

5.3. Processing Rules for Creating Data Objects

The common processing rules specified earlier MUST also be followed (see Section 3).

5.3.1. Multiple <CreateItem> Elements

One <**Create**> element can contain multiple <**CreateItem**> elements. The following rules specify how those must be supported and handled:

1. A WSP MUST support one <**CreateItem**> element inside a <**Create**> and SHOULD support multiple. If a WSP supports only one <**CreateItem**> element inside a <**Create**> and the <**Create**> contains multiple <**CreateItem**> elements, the processing of the whole <**Create**> MUST fail and a status code indicating failure MUST be returned in the response. A more detailed status code with the value `NoMultipleAllowed` SHOULD be used in addition to the top level status code. If a WSP supports multiple <**CreateItem**> elements inside a <**Create**>, it MAY register the urn:liberty:dst:multipleCreateItems discovery option keyword.

2. If the processing of a <**CreateItem**> fails even partly due to some reason, depending on the service and/or a WSP either the processing of the whole <**Create**> MUST fail or a WSP MUST try to achieve partial success. The top level status code `Failed` or `Partial` MUST be used to indicate the failure (complete or partial) and a more detailed second level status code SHOULD be used to indicate the reason for failing to completely process the failed <**Create**> element. Furthermore, the `ref` XML attribute of the <**Status**> element SHOULD carry the value of the `itemID`of the failed <**CreateItem**> element and in partial success cases it MUST carry this value. The modifications made based on already processed <**CreateItem**> elements of the <**Create**> MUST be rolled back in case of a complete failure. A WSP MUST NOT support multiple <**CreateItem**> elements inside one <**Create**>, if it cannot roll back and partial failure is not allowed.

3. When multiple <**CreateItem**> elements inside one <**Create**> element are supported and partial success is allowed, a WSC MUST use the `itemID` XML attribute in each <**CreateItem**> element so that a WSP can identify the failed parts, when it is returning status information for a partial success.

5.3.2. Only One Type of Data Object per <CreateItem>

With one <**CreateItem**> element a WSC can add only one type of data objects, but the amount of object may vary.

1. A WSP MUST support multiple data objects of the same type inside the <**NewData**> element of a <**CreateItem**> element, if the service can have multiple objects of that type, unless otherwise specified in a service specification. If a data object inside a <**NewData**> element is not of the type specified by the `objectType` XML attribute of the <**CreateItem**> containing the <**NewData**> element, the processing of that <**CreateItem**> MUST fail and second level status code `ObjectTypeMismatch` SHOULD be used. If the data inside a <**NewData**> is otherwise unacceptable to a WSP, the processing of the <**CreateItem**> MUST fail and second level status code `InvalidData` SHOULD be used unless some better service or object type specific status code has been defined in the service specification or in this specification. A data object might contain an <**Extension**> element, which has some data not specified in the service specification. A WSP might not support extensions and not accept that data. This SHOULD be indicated with the second level status code `ExtensionNotSupported`.

2. If there is no **<NewData>** element inside a **<CreateItem>**, an empty data object of the type specified by the `objectType` XML attribute MUST be created unless service specification requires that a object always has some data, e.g., an identifier created by a WSC to be used to access that specific object instead of other objects of the same type. If a **<NewData>** element is required inside a **<CreateItem>** element and it is missing, the processing of that **<CreateItem>** MUST fail and second level status code `MissingNewData` should be used to indicate this.

5.3.3. Handling `commonAttributes` and `leafAttributes` upon Creation

The common XML attributes belonging to the XML attribute groups `commonAttributes` and `leafAttributes` are mainly supposed to be written by the WSP hosting the data service. There are some additional rules for handling these common XML attributes when data objects are created.

1. When any of the `ACC`, `modifier`, `ACCTime` or `modificationTime` XML attributes is used in a resource, the WSP hosting the data service MUST keep their values up to date. When a data object is created, the `modifier` XML attribute MUST contain the ProviderID of the creator or have no value, and the `modificationTime` MUST define the time of the creation or have no value. The `ACC` MUST define the XML attribute collection context of the current value of a data element or have no value and the `ACCTime` MUST define the time, when the value of the `ACC` was defined or have no value.

2. If the **<NewData>** contains `modifier`, `modificationTime` or `ACCTime` XML attributes for any data element, the WSP MUST ignore these and update the values based on other information than those XML attributes inside the **<NewData>** provided by the WSC. If the `ACC` XML attribute is included for any data element, the WSP MAY accept it, depending on how much it trusts the requesting service provider. The WSP MAY also accept the `id` XML attribute provided inside the **<NewData>** and some services MAY require that the `id` XML attribute MUST be provided by the requesting WSC.

3. The `id` XML attribute MUST NOT be used as a global unique identifier. The value MUST be chosen so that it works only as unique identifier inside the conceptual XML document.

4. When a data object is created based on a **<Create>** request, the values of the `modificationTime` XML attributes written by the WSP hosting the data service MAY be same for all elements of created object, but there is no guarantee that they will be exactly the same. When the `modificationTime` XML attribute is used in container elements, the time of a modification MUST be propagated to all ancestor elements of the modified element all the way up to the root element. So the root element has always the latest modification time.

5.3.4. WSC Might Mot Be Allowed to Add Certain Data or Any Data

When a WSP processes a **<CreateItem>**, it MUST check, whether the resource owner (for example, the Principal) has given consent to the requestor to create the data. To be able to check WSC-specific access rights, the WSP MUST authenticate the WSC (see [LibertySecMech]). If the consent check fails for any part of the requested data, the WSP MUST NOT create data requested in the **<CreateItem>** element, even when such consent is missing only for some subelement or XML attribute. The WSP MAY try to get consent from the Principal while processing the request perhaps using an interaction service (for more information see [LibertyInteract]). The processing of a **<CreateItem>** element MUST fail, if the creation of the data object was not allowed. The second level status code `ActionNotAuthorized` MAY be used, if it is considered that the privacy of the owner of the resource is not compromised. A WSP might check the access rights at a higher level, before getting to DST processing and MAY return an ID-* Fault Message [LibertySOAPBinding] and not process the **<Create>** element at all, if the requesting WSC is not allowed to create data objects.

5.3.5. WSP May Place Some Restrictions for the data It Is Hosting

1. The schemata for different data services may have some elements for which there is not an exact upper limit on how many can exist. For practical reasons, implementations may set some limits. If a request tries to add more elements than a WSP supports, the WSP will not accept the new element(s) and the processing of the **<CreateItem>** element MUST fail. The WSP should use a second level status code `NoMoreElements` to indicate this specific case. If a WSC tries to add more data object than a WSP supports, the processing of the **<CreateItem>** element MUST fail and the second level status code `NoMoreObjects` to indicate this. If only one data object of the type specified by the `objectType` is allowed and a WSC tries to create it although it already exists, the correct second level status code is `ExistsAlready`.

2. The schemata for different data services may not specify the length of elements and XML attributes especially in the case of strings. If a request tries to add longer values for data elements or XML attributes than a WSP supports, the WSP may not accept the data and the processing of the **<CreateItem>** element will fail. The WSP should use a second level status code `DataTooLong` to indicate this.

6. Deleting Data Objects

A WSC can delete existing data objects, when a service supports multiple data objects of the same type.

6.1. <Delete> Element

The <**Delete**> element is used to delete existing data objects, not data inside a data object, but whole objects including the contained data. If only the data inside an object should be deleted, a WSC must use <**Modify**> for it.

The data objects to be deleted are referred to either using the `predefined` XML attribute or the `objectType` XML attribute and the <**Select**> element in the <**DeleteItem**> element. Concurrent updates are handled using the `notChangedSince` XML attribute inside the <**DeleteItem**> element. If the data has been modified since the time specified by the `notChangedSince` XML attribute, the deletion MUST NOT be done.

```
<xs:complexType name="DeleteItemBaseType">
  <xs:attributeGroup ref="dst:selectQualif"/>
  <xs:attribute name="notChangedSince" use="optional" type="xs:dateTime"/>
  <xs:attribute name="id" use="optional" type="xs:ID"/>
  <xs:attribute ref="lu:itemID" use="optional"/>
</xs:complexType>
<xs:complexType name="DeleteResponseType">
  <xs:complexContent>
    <xs:extension base="lu:ResponseType"/>
  </xs:complexContent>
</xs:complexType>
```

Figure 14. Utility Schema for Delete

```
<xs:complexType name="DeleteType">
  <xs:complexContent>
    <xs:extension base="dst:RequestType">
      <xs:sequence>
        <xs:element ref="dstref:DeleteItem" minOccurs="1" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:element name="DeleteItem" type="dstref:DeleteItemType"/>
<xs:complexType name="DeleteItemType">
  <xs:complexContent>
    <xs:extension base="dst:DeleteItemBaseType">
      <xs:sequence>
        <xs:element ref="dstref:Select" minOccurs="0" maxOccurs="1"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="DeleteResponseType">
  <xs:complexContent>
    <xs:extension base="lu:ResponseType"/>
  </xs:complexContent>
</xs:complexType>
```

Figure 15. Reference Model for Delete

6.2. <DeleteResponse> Element

The <**DeleteResponse**> element contains mainly the mandatory <**Status**> element. No time stamp is returned as the data does not exist after processing the request.

6.3. Processing Rules for Deletion

The common processing rules specified earlier MUST also be followed (see Section 3).

6.3.1. Supporting Multiple <DeleteItem> Elements

One **<Delete>** element can contain multiple **<DeleteItem>** elements. The following rules specify how those must be supported and handled:

1. A WSP MUST support one **<DeleteItem>** element inside a **<Delete>** and SHOULD support multiple. If a WSP supports only one **<DeleteItem>** element inside a **<Delete>** and the **<Delete>** contains multiple **<DeleteItem>** elements, the processing of the whole **<Delete>** MUST fail and a status code indicating failure MUST be returned in the response. A more detailed status code with the value `NoMultipleAllowed` SHOULD be used in addition to the top level status code. If a WSP supports multiple **<DeleteItem>** elements inside a **<Delete>**, it MAY register the urn:liberty:dst:multipleDeleteItems discovery option keyword.

2. If the processing of a **<DeleteItem>** fails even partly due to some reason, depending on the service and/or a WSP either the processing of the whole **<Delete>** MUST fail or a WSP MUST try to achieve partial success. The top level status code `Failed` or `Partial` MUST be used to indicate the failure (complete or partial) and a more detailed second level status code SHOULD be used to indicate the reason for failing to completely process the failed **<Delete>** element. Furthermore, the `ref` XML attribute of the **<Status>** element SHOULD carry the value of the `itemID` of the failed **<DeleteItem>** element and in partial success cases it MUST carry this value. The deletions made based on already processed **<DeleteItem>** elements of the **<Delete>** MUST be rolled back in case of a complete failure. A WSP MUST NOT support multiple **<DeleteItem>** elements inside one **<Delete>**, if it cannot roll back and partial failure is not allowed.

3. When multiple **<DeleteItem>** elements inside one **<Delete>** element are supported and partial success is allowed, a WSC MUST use the `itemID` XML attribute in each **<DeleteItem>** element so that a WSP can identify the failed parts, when it is returning status information for a partial success.

6.3.2. Only One Type of Data Object May Be Deleted with One <DeleteItem>

With one **<DeleteItem>** element a WSC can delete only one type of data objects unless `predefined` XML attribute is used, but the amount of object may vary.

1. All data objects matching the selection criteria given in a **<DeleteItem>**, either `predefined` XML attribute or `objectType` XML attribute and **<Select>** element, MUST be deleted. If all matching can not be deleted, the processing of that **<DeleteItem>** MUST fail and appropriate second level status code should be used to indicate the reason. If a **<DeleteItem>** fails, a WSP MUST NOT delete any data based on it.

2. If there is no **<Select>** element inside a **<DeleteItem>**, all data objects of the type specified by the `objectType` XML attribute MUST be deleted. A service specification may require that **<Select>** element is always used, when the `predefined` XML attribute is not used.

6.3.3. Avoiding Deletion of Data if It Has Changed In-between

A WSC might want to avoid deleting data, if someone else has changed it in-between.

When the `notChangedSince` XML attribute is present, the deletions specified by a **<DeleteItem>** element MUST NOT be made, if any part of the data to be deleted has changed since the time specified by the `notChangedSince` XML attribute. The second level status code `ModifiedSince` MUST be used to indicate that the deletion was not done because the data has been modified since the time specified by the `notChangedSince` XML attribute. If a WSP does not support processing of this XML attribute properly, it MUST NOT make any changes and it MUST return the second level status code `ChangeHistoryNotSupported`. If a WSP supports this `notChangedSince` XML attribute, it MUST also support the `changedSince` XML attribute of the **<QueryItem>** element and `notChangedSince` XML attribute of the **<ModifyItem>**.

6.3.4. WSC Might Not Be Allowed to Delete Certain or Any Data

When a WSP processes a **<DeleteItem>**, it MUST check, whether the resource owner (for example, the Principal) has given consent to the requestor to delete the data. To be able to check WSC-specific access rights, the WSP MUST authenticate the WSC (see [LibertySecMech]). If the consent check fails for any part of the data requested to be deleted, the WSP MUST NOT delete data requested in the **<DeleteItem>** element, even when such consent is missing only for some subelement or XML attribute. The WSP MAY try to get consent from the Principal while processing the request, for example, using an interaction service (for more information see [LibertyInteract]). The processing of a **<DeleteItem>** element MUST fail, if the deletion of a data object was not allowed. The second level status code `ActionNotAuthorized` MAY be used, if it is considered that the privacy of the owner of the resource is not compromised. A WSP might check the access rights at a higher level, before getting to DST processing and MAY return an ID-* Fault Message [LibertySOAPBinding] and not process the **<Delete>** element at all, if the requesting WSC is not allowed to delete data objects.

7. Modifying Data

The data objects stored by a data service can be modified. Usually the Principal can make these modifications directly at the data service using the provided user interface, but these modifications may also be made by other service providers using the <**Modify**> element. It is not possible to create or delete data objects with the <**Modify**>, just change of existing data objects.

7.1. <Modify> Element

The <**Modify**> element has two types of sub-elements.

- <**ModifyItem**> elements specify which data elements of the specified resource should be modified and how.
- <**ResultQuery**> elements can be included, when a WSC wants, for example, to get back data related to the modifications it just made.

The `objectType` XML attribute and the <**Select**> element inside a <**ModifyItem**> element specifies the data this modification should affect. The <**Select**> element is not needed when a resource in a data service has only one data object of the type specified with the `objectType` XML attribute and the whole content of that data object is modified. If a data service supports only one `objectType`, this XML attribute may be omitted.

The <**NewData**> subelement of <**ModifyItem**> defines the new values for the data addressed by the `objectType` XML attribute and the <**Select**> element. The new values, specified inside the <**NewData**> element, replace any existing selected data, if the `overrideAllowed` XML attribute of the <**ModifyItem**> element is set to `True`.

If the <**NewData**> element does not exist or is empty, it means than the selected current data values should be removed. Note that whole data object can be deleted only with a separate <**Delete**> message, not with <**Modify**>. The default value for the `overrideAllowed` XML attribute is `False`, which means that the <**ModifyItem**> is only allowed to add new data to a data object, not to remove or replace existing data of a data object.

The `notChangedSince` XML attribute is used to handle concurrent updates. When the `notChangedSince` XML attribute is present, a modification is allowed to be done only if the data to be modified has not changed since the time specified by the value of the `notChangedSince` XML attribute.

The <**ModifyItem**> element MUST also have the `itemID` XML attribute, when multiple <**ModifyItem**> elements are included in one <**Modify**> element and partial failure is allowed so that failed parts can be identified.

A <**Modify**> may include <**ResultQuery**> elements, if a WSC wants to get back data it is just modifying to, for example, find out the details, was all the new data accepted, or get back possible metadata a WSP might have added to the modified data.

```
<xs:attributeGroup name="ModifyItemAttributeGroup">
  <xs:attributeGroup ref="dst:selectQualif"/>
  <xs:attribute name="notChangedSince" use="optional" type="xs:dateTime"/>
  <xs:attribute name="overrideAllowed" use="optional" type="xs:boolean" default="0"/>
  <xs:attribute name="id" use="optional" type="xs:ID"/>
  <xs:attribute ref="lu:itemID" use="optional"/>
</xs:attributeGroup>
```

Figure 16. XML Attributes for Modify

```
<xs:complexType name="ModifyType">
  <xs:complexContent>
    <xs:extension base="dst:RequestType">
      <xs:sequence>
        <xs:element ref="dstref:ModifyItem" minOccurs="1" maxOccurs="unbounded"/>
        <xs:element ref="dstref:ResultQuery" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:element name="ModifyItem" type="dstref:ModifyItemType"/>
<xs:complexType name="ModifyItemType">
  <xs:sequence>
    <xs:element ref="dstref:Select" minOccurs="0" maxOccurs="1"/>
    <xs:element ref="dstref:NewData" minOccurs="0" maxOccurs="1"/>
  </xs:sequence>
  <xs:attributeGroup ref="dst:ModifyItemAttributeGroup"/>
</xs:complexType>
<xs:complexType name="ModifyResponseType">
  <xs:complexContent>
    <xs:extension base="dstref:DataResponseType"/>
  </xs:complexContent>
</xs:complexType>
```

Figure 17. Reference Model for Modify

7.2. <ModifyResponse> Element

The <**ModifyResponse**> element contains the <**Status**> element, which describes whether or not the requested modification succeeded. There is also a possible time stamp XML attribute, which provides a time value that can be used later to check whether there have been any changes since this modification, and an `itemIDRef` XML attribute to map the <**ModifyResponse**> elements to the <**Modify**> elements in the request.

A <**ModifyResponse**> may also contain <**ItemData**> elements which contain data requested with <**ResultQuery**> elements. One <**ItemData**> element MUST NOT contain more data than requested with one <**ResultQuery**> element. Note that a WSP MAY return data using the <**ItemData**> element even when a WSC did not ask for it, if a WSP thinks that a WSC needs that data, e.g., to access it later on.

7.3. Processing Rules for Modifications

The common processing rules specified earlier MUST also be followed (see Section 3).

7.3.1. Multiple <ModifyItem> Elements

1. A WSP MUST support one <**ModifyItem**> element inside a <**Modify**> and SHOULD support multiple. If the <**Modify**> contains multiple <**ModifyItem**> elements and the WSP supports only one <**ModifyItem**> element inside a <**Modify**>, the processing of the whole <**Modify**> MUST fail and a status code indicating failure MUST be returned in the response. The value `NoMultipleAllowed` SHOULD be used for the second level status code. If a WSP supports multiple <**ModifyItem**> element inside a <**Modify**>, it MAY register the urn:liberty:dst:multipleModifyItem discovery option keyword.

2. If the processing of a **<ModifyItem>** fails even partly due to some reason, depending on the service and/or a WSP either the processing of the whole **<Modify>** MUST fail or a WSP MUST try to achieve partial success. The top level status code `Failed` or `Partial` MUST be used to indicate the failure (complete or partial) and a more detailed second level status code SHOULD be used to indicate the reason for failing to completely process the failed **<Modify>** element. Furthermore, the `ref` XML attribute of the **<Status>** element SHOULD carry the value of the `itemID` of the failed **<ModifyItem>** element and in partial success cases it MUST carry this value. The modifications made based on already processed **<ModifyItem>** elements of the **<Modify>** MUST be rolled back in case of a complete failure. A WSP MUST NOT support multiple **<ModifyItem>** elements inside one **<Modify>**, if it cannot roll back and partial failure is not allowed.

3. When multiple **<ModifyItem>** elements inside one **<Modify>** element are supported and partial success is allowed, a WSC MUST use the `itemID` XML attribute in each **<ModifyItem>** element so that a WSP can identify the failed parts, when it is returning status information for a partial success.

7.3.2. What Exactly Is Modified

What is modified and how depends on a number of parameters including the value of the **<Select>** element, the content of the provided **<NewData>** element, the value of the `overrideAllowed` XML attribute, and the current content of the underlying conceptual XML document.

1. When adding new data, the **<Select>** element will point in the conceptual XML document to an element which does not exist yet. The new element is added as a result of processing the **<ModifyItem>** element. In such cases, when the ancestor elements of the new element do not exist either, they MUST be added as part of processing of the **<ModifyItem>** element so that processing could be successful.

2. If the **<Select>** points to multiple places and there is a **<NewData>** element with new values, the processing of the **<ModifyItem>** MUST fail because it is not clear where to store the new data. If there is no **<NewData>** element and the `overrideAllowed` XML attribute is set to `True`, then the processing of **<ModifyItem>** can continue normally, because it is acceptable to delete multiple data elements at once (for example, all AddressCards).

3. When the `overrideAllowed` is set to `False` or is missing, the **<NewData>** element MUST be present as new data should be added. If the **<NewData>** element is missing in this case, the processing of the **<ModifyItem>** MUST fail and the second level status code `MissingNewDataElement` SHOULD be returned in addition to top level status code.

4. When there is the **<NewData>** element with new values and the **<Select>** points to existing information, the processing of the **<ModifyItem>** MUST fail, if the `overrideAllowed` XML attribute is not set to `True`. When the `overrideAllowed` XML attribute does not exist or is set to `False`, the new data in the **<NewData>** element can only be accepted in two cases: either there is no existing element to which the **<Select>** points, or there can be multiple data elements of the same type. This means that, if the **<Select>** points to an existing container element, which has a subelement, and only one such container element can exist, the **<ModifyItem>** MUST fail, even if the only subelement the container element has inside the **<NewData>** does not yet exist in the conceptual XML document. The second level status code `ExistsAlready` SHOULD be used to indicate in details the reason for the failure in addition to the top level status code. The lack of those other sub-elements inside the **<NewData>** means that they should be removed, which is only possible when `overrideAllowed` XML attribute equals to `True`.

5. When there can be multiple elements of the same type, the addition of a new element MUST fail, if there exists already an element of same type have the same value of the distinguishing part. In the case of a personal profile service, adding a new **<AddressCard>** element MUST fail, if there already exists an **<AddressCard>** element which has an `id` XML attribute of the same value as the provided new **<AddressCard>** element. The second level status code `ExistsAlready` SHOULD also be used to indicate the detailed reason for failure.

6. When all or some of the data inside the **<NewData>** element is not supported by the WSP, or the provided data is not valid, the processing of the whole **<ModifyItem>** SHOULD fail and second level status code `InvalidData` SHOULD be returned in the response.

7. When the **<ModifyItem>** element tries to extend the service either by pointing to a new data type behind an **<Extension>** element with the **<Select>** element, or having new sub-elements under an **<Extension>** element inside the **<NewData>** element and the WSP does not support extension in general or for the requesting party, it SHOULD be indicated in the response message with the second level status code `ExtensionNotSupported`.

8. When the WSP supports extensions, but does not accept the content of the **<Select>** or **<NewData>**, then second level status codes `InvalidSelect` and `InvalidData` SHOULD be used as already described.

7.3.3. Handling `commonAttributes` and `leafAttributes` in Modify

The common XML attributes belonging to the XML attribute groups `commonAttributes` and `leafAttributes` are mainly supposed to be written by the WSP hosting the data service. There are some additional rules for handling these common XML attributes in case of modifications.

1. When any of the `ACC`, `modifier`, `ACCTime` or `modificationTime` XML attributes is used in a resource, the WSP hosting the data service MUST keep their values up to date. When data is modified, the `modifier` MUST contain the ProviderID of the modifier or have no value, and the `modificationTime` MUST define the time of the modification or have no value. The `ACC` MUST define the XML attribute collection context of the current value of a data element or have no value and the `ACCTime` MUST define the time, when the current value of the `ACC` was defined or have no value.

2. If the **<NewData>** contains `modifier`, `modificationTime` or `ACCTime` XML attributes for any data element, the WSP MUST ignore these and update the values based on other information than those XML attributes inside the **<NewData>** provided by the WSC. If the `ACC` XML attribute is included for any data element, the WSP MAY accept it, depending on how much it trusts the requesting service provider. The WSP MAY also accept the `id` XML attribute provided inside the **<NewData>** and some services MAY require that the `id` XML attribute MUST be provided by the requesting service provider.

3. The `id` XML attribute MUST NOT be used as a global unique identifier. The value MUST be chosen so that it works only as unique identifier inside the conceptual XML document, and the value of the `id` XML attribute SHOULD be kept the same even if the element is otherwise modified. A WSP MAY not even allow changing the value of the `id` XML attribute or any other XML attribute used to distinguish elements with the same name from each other.

4. When data is modified based on the **<Modify>** request, the values of the `modificationTime` XML attributes written by the WSP hosting the data service MAY be same for all inserted and updated elements, but there is no guarantee that they will be exactly the same. When the `modificationTime` XML attribute is used by a data service, the WSP MUST keep it up to date to indicate the time of the latest modification of an element and update it, when ever a modification is done either using the **<Modify>** request or some other way. When the `modificationTime` XML attribute is used in container elements, the time of a modification MUST be propagated to all ancestor elements of the modified element all the way up to the root element.

7.3.4. Accounting for Concurrent Updates

Accounting for concurrent updates is handled using the `notChangedSince` XML attribute inside the **<ModifyItem>** element.

1. When the `notChangedSince` XML attribute is present, the modifications specified by the **<ModifyItem>** element MUST NOT be made, if any part of the data to be modified has changed since the time specified by the `notChangedSince` XML attribute.

2. The second level status code `ModifiedSince` MUST be used to indicate that the modification was not done because the data has been modified since the time specified by the `notChangedSince` XML attribute. If a WSP does not support processing of this XML attribute properly, it MUST NOT make any changes and it MUST return the second level status code `ChangeHistoryNotSupported`. If a WSP supports this `notChangedSince` XML attribute, it MUST also support the `changedSince` XML attribute of the <**QueryItem**> element.

7.3.5. WSC Might Not Be Allowed to Make Only Certain or Any Modifications

When a WSP processes the <**ModifyItem**>, it MUST check, whether the resource owner (for example, the Principal) has given consent to the requestor to modify the data. To be able to check WSC-specific access rights, the WSP MUST authenticate the WSC (see [LibertySecMech]). If the consent check fails for any part of the requested data, the WSP MUST NOT make the modifications requested in the <**ModifyItem**> element, even when such consent is missing only for some subelement or XML attribute. The WSP MAY try to get consent from the Principal while processing the request perhaps using an interaction service (for more information see [LibertyInteract]). The processing of the <**ModifyItem**> element MUST fail, if the modification was not allowed. The second level status code `ActionNotAuthorized` MAY be used, if it is considered that the privacy of the owner of the resource is not compromised. A WSP might check the access rights at a higher level, before getting to DST processing and MAY return an ID-* Fault Message [LibertySOAPBinding] and not process the <**Modify**> element at all, if the requesting WSC is not allowed to modify the data.</**para**>

7.3.6. WSP May Impose Some Restrictions for the Data It Is Hosting

1. The schemata for different data services may have some elements for which there is not an exact upper limit on how many can exist. For practical reasons, implementations may set some limits. If a request tries to add more elements than a WSP supports, the WSP will not accept the new element(s) and the processing of the <**ModifyItem**> element MUST fail. The WSP should use a second level status code `NoMoreElements` to indicate this specific case.

2. The schemata for different data services may not specify the length of elements and XML attributes especially in the case of strings. The WSP may also have limitations of this kind. If a request tries to add longer data elements or XML attributes than a WSP supports, the WSP may not accept the data and the processing of the <**ModifyItem**> element will fail. The WSP should use a second level status code `DataTooLong` to indicate this specific case.

7.4. Examples of Modifications

This example adds a home address to the personal profile of a Principal:

```
<hp:Modify xmlns:hp="urn:liberty:hp:2005-07">
  <hp:ModifyItem>
    <hp:Select>/hp:HP/hp:AddressCard</hp: Select>
    <hp:NewData>
      <hp:AddressCard id='98123'>
        <hp:AddressType>
          urn:liberty:hp:addrType:home
        </hp:AddressType>
        <hp:Address>
          <hp:PostalAddress>
            c/o Carolyn Lewis$2378 Madrona Beach Way North
          </hp:PostalAddress>
          <hp:PostalCode>98503-2341</hp :PostalCode>
          <hp:L>Olympia</hp:L>
          <hp:ST>wa</hp:ST>
          <hp:C>us</hp:C>
        </hp:Address>
      </hp:AddressCard>
    </hp:NewData>
  </hp:ModifyItem>
</hp:Modify>
```

The following example replaces the current home address with a new home address in the personal profile of a Principal. Please note that this request will fail if there are two or more home addresses in the profile, because it is not clear in this request, which of those addressed should be replaced by this address. In such a case the `id` XML attribute should be used to explicitly point which of the addresses should be changed.

```
<hp:Modify xmlns:hp="urn:liberty:hp:2005-07">
  <hp:ModifyItem overrideAllowed="True">
     <hp:Select>
       /hp:HP/hp:AddressCard
         [hp:AddressType='urn:liberty:id-sis-hp:addrType:home']
     </hp:Select>
     <hp:NewData>
       <hp:AddressCard id="98123">
         <hp:AddressType>
           urn:liberty:id-sis-hp:addrType:home
         </hp:AddressType>
         <hp:Address>
           <hp:PostalAddress>
             c/o Carolyn Lewis$2378 Madrona Beach Way
           </hp:PostalAddress>
           <hp:PostalCode>98503-2342</hp:PostalCode>
           <hp:L>Olympia</hp:L>
           <hp:ST>wa</hp:ST>
           <hp:C>us</hp:C>
         </hp:Address>
       </hp:AddressCard>
     </hp:NewData>
  </hp:ModifyItem>
</hp:Modify>
```

This example replaces the current address identified by an `id` of '98123' with a new home address, if that address has not been modified since 12:40:01 21th January 2003 UTC.

```
<hp:Modify xmlns:hp="urn:liberty:hp:2005-07">
  <hp:ModifyItem
     notChangedSince="2003-01-21T12:40:01Z"
     overrideAllowed="True">
    <hp:Select>/hp:HP/hp:AddressCard[@hp:id='98123']</hp:Select>
    <hp:NewData>
      <hp:AddressCard id="98123">
        <hp:AddressType>
          urn:liberty:id-sis-hp:addrType:home
        </hp:AddressType>
        <hp:Address>
          <hp:PostalAddress>
            c/o Carolyn Lewis$2378 Madrona Beach Way South
          </hp:PostalAddress>
          <hp:PostalCode>98503-2398</hp:PostalCode>
          <hp:L>Olympia</hp:L>
          <hp:ST>wa</hp:ST>
          <hp:C>us</hp:C>
        </hp:Address>
      </hp:AddressCard>
    </hp:NewData>
  </hp:ModifyItem>
</hp:Modify>
```

The following example adds another home address to the personal profile of a Principal. An `id` is provided for the new address.

```
<hp:Modify xmlns:hp="urn:liberty:hp:2005-07">
  <hp:ModifyItem>
    <hp:Select>
      /hp:HP/hp:AddressCard
        [hp:AddressType='urn:liberty:id-sis-hp:addrType:home']
    </hp:Select>
    <hp:NewData>
      <hp:AddressCard id="12398">
        <hp:AddressType>
          urn:liberty:id-sis-hp:addrType:home
        </hp:AddressType>
        <hp:Address>
          <hp:PostalAddress>1234 Beach Way</hp:PostalAddress>
          <hp:PostalCode>98765-1234</hp:PostalCode>
          <hp:L>Olympia</hp:L>
          <hp:ST>wa</hp:ST>
          <hp:C>us</hp:C>
        </hp:Address>
      </hp:AddressCard>
    </hp:NewData>
  </hp:ModifyItem>
</hp:Modify>
```

The following example removes all current home addresses from the personal profile of a Principal:

```
<hp:Modify xmlns:hp="urn:liberty:hp:2005-07">
  <hp:ModifyItem overrideAllowed="True">
    <hp:Select>
      /hp:HP/hp:AddressCard
        [hp:AddressType='urn:liberty:id-sis-hp:addrType:home']
    </hp:Select>
  </hp:ModifyItem>
</hp:Modify>
```

The response for a valid <**Modify**> is as follows:

```
<hp:ModifyResponse
   xmlns:hp="urn:liberty:hp:2005-07"
   timeStamp="2003-03-23T03:40:00Z">
  <hp:Status code="OK"/>
</hp:ModifyResponse>
```

8. WSF-1.1 Compatibility

This version (2.1) of DST was designed to work well with ID-WSF 2.0 specification family. Since it is a major version upgrade, a decision was made to break the ID-WSF 1.1 compatibility, mainly by elimination of the **<ResourceIDs>** (see also Section 3.6).

However, the two ID-WSF versions remain broadly compatible. [LibertyDisco], Section 10 "ID-WSF 1.x Resource Offering conveyed in an EPR" provides a method for constructing **<ResourceID>**s from credentials as well as making credentials and end points given knowledge of the `ResourceID`. The frame work version header, see [LibertySOAPBinding] allows simultaneous support to be implemented at run time.

9. Actions

When SOAP action names are need, they SHOULD be formed by appending to *service type* one of the Request names, i.e., `Create`, `Delete`, `Query`, `Modify`, etc.

Example

```
urn:liberty:id-sis-dap:2005-10:dst-2.1:Query
```

10. Checklist for Service Specifications

The following is a checklist of issues which should be addressed by individual service type specifications. Such specifications should always state which optional features of the DST they support, in addition to defining more general things such as discovery option keywords and the `SelectType` XML type used by the service type. A service specification should complete this list with the specific values and statements required by the specification.

For optional features, the language specified by [RFC2119] MUST be used to define whether these features are available for implementations and deployments. For example, specifying that a feature 'MAY' be implemented by a WSP means that WSPs may or may not support the feature, and that WSCs should be ready to handle both cases.

Default feature support policy is that all features, unless expressly waived by service specification, MUST be supported, but each feature MAY be disabled administratively or by configuration in a deployment (e.g., to provide read or write only service).

1. Specify service type. Specify namespaces for the service if different from service type.

2. Provide definition service schema including the methods as elements based on DST types. A service need not define every possible method supported by DST and it may define additional methods supported by service specific schema. The service may also rename some of the methods. If it does rename, it MUST state which DST method corresponds to the renamed method. There can be several service methods that map to one DST method.

3. Enumerate object types

4. Describe `AppDataType` and its contents. The description can come in form of XML schema, or the description can simply describe the contents of the string that is to appear in elements derived from `AppDataType`, i.e., <**NewData**>, <**Data**>, and <**ItemData**>. The data description may make allowance for different object types.

5. Describe `SelectType` and how it applies to various types of objects. If selects can not be described as a string, e.g., XPath can, the service may want to redefine the type using xs:redefine.

   It is possible that different query language or dialect is applied depending on which object type is being queried. If so, the service specification MUST resolve how to represent the different languages using one `SelectType`

6. Describe `TestOpType`, considering how to test all object types supported by the specification. It is possible that different test language or dialect is applied depending on which object type is being tested. If so, the service specification MUST resolve how to represent the different languages using one `TestOpType`

7. Describe `SortType`.

8. Enumerate Methods and state the required level of support. The default set of methods is <**Create**>, <**Query**>, <**Modify**>, and <**Delete**>.

   Default method support policy is as follows

   a. All methods MUST be supported, but each method MAY be disabled administratively or by configuration in a deployment (e.g. to provide read only or write only service).

   b. If queries are disabled or access control makes it implausible that they succeed, discovery option keyword urn:liberty:dst:noQuery MUST be registered.

   c. If creates are disabled or access control makes it implausible that they succeed, discovery option keyword urn:liberty:dst:noCreate MUST be registered.

   d. If deletes are disabled or access control makes it implausible that they succeed, discovery option keyword urn:liberty:dst:noDelete MUST be registered.

e. If modifies are disabled or access control makes it implausible that they succeed, discovery option keyword urn:liberty:dst:noModify MUST be registered.

9. The discovery option keywords (see [LibertyDisco]) can either be listed with semantics here, or via a reference to the correct chapter in the specification. Please note that the DST defines the following discovery option keywords and the service specification must list which of these the service may use:

```
urn:liberty:dst:allPaths
urn:liberty:dst:can:extend
urn:liberty:dst:changeHistorySupported
urn:liberty:dst:contingentQueryItems
urn:liberty:dst:extend
urn:liberty:dst:fullXPath
urn:liberty:dst:multipleCreateItems
urn:liberty:dst:multipleDeleteItems
urn:liberty:dst:multipleModifyItem
urn:liberty:dst:multipleQueryItems
urn:liberty:dst:multipleResources
urn:liberty:dst:noQuery
urn:liberty:dst:noCreate
urn:liberty:dst:noDelete
urn:liberty:dst:noModify
urn:liberty:dst:noPagination
urn:liberty:dst:noSorting
urn:liberty:dst:noStatic
```

10. Element uniqueness. State here how elements with the same name are distinguished from each other. For example, the `id` XML attribute is used for **<AddressCard>** and **<MsgContact>** elements, `xml:lang` and `script` XML attributes are used for localized elements, etc. Element uniqueness MUST consider different object types.

11. Extension support. State whether extension is supported and if so, describe this support. A reference to the specification chapter defining this can be given. For example, "New elements and discovery option keywords MAY be defined, see chapter Y.X for more details."

    Extensions support should discuss both data extension and protocol extension, including **<Extension>** elements request and response messages.

    The default policy for protocol extension is that mutually consenting WSC and WSP MAY use extension points for implementation dependent purposes. Extension points that can be thus used are

    a. XML any extension points contained in **<Extension>** elements that are present in various protocol messages, provided that the extension elements are namespace qualified.

    b. If `SelectType`, `TestOpType`, or `SortType` is designated as unused by the service specification, then it MAY be used for extension, provided that the extension data is

        a. in URI format and use an assigned domain name as a component of the URI to ensure that extensions do no collide with each other.

        b. Namespace qualified XML document

12. Statement of optionality of query features (and their manifestation on discovery option keywords, see above):

    a. Support testing

    b. Support **<ResultQuery>**

c. Support sorting

d. Support pagination of results

e. Support static sets in pagination

f. Support multiple <**Query**> elements

g. Support multiple <**QueryItem**> elements

h. Support multiple <**TestItem**> elements

i. Support `changedSince` (and which formats) in <**ResultQuery**> and <**QueryItem**>

j. Support `includeCommonAttributes`

13. Statement of optionality of create features (and their manifestation on discovery option keywords, see above):

a. Support multiple <**Create**> elements

14. Statement of optionality of delete features (and their manifestation on discovery option keywords, see above):

a. Support multiple <**Delete**> elements

15. Statement of optionality of modify features (and their manifestation on discovery option keywords, see above):

a. Support multiple <**Modify**> elements

b. Support multiple <**ModifyItem**> elements

c. Support partial success. If multiple <**ModifyItem**> elements are supported, is partial success supported or are only atomic modifications allowed?

d. Support `notChangedSince`

11. Schemata

11.1. DST Reference Model Schema

The formal schema for the reference model follows.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
   targetNamespace="urn:liberty:dst:2006-08:ref"
   xmlns:dstref="urn:liberty:dst:2006-08:ref"
   xmlns:dst="urn:liberty:dst:2006-08"
   xmlns:lu="urn:liberty:util:2006-08"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   elementFormDefault="qualified"
   attributeFormDefault="unqualified">
  <xs:import namespace="urn:liberty:dst:2006-08"
     schemaLocation="liberty-idwsf-dst-v2.1.xsd"/>
  <xs:import namespace="urn:liberty:util:2006-08"
     schemaLocation="liberty-idwsf-utility-v2.0.xsd"/>
<!--sec(methods)-->
  <xs:element name="Create" type="dstref:CreateType"/>
  <xs:element name="CreateResponse" type="dstref:CreateResponseType"/>
  <xs:element name="Query" type="dstref:QueryType"/>
  <xs:element name="QueryResponse" type="dstref:QueryResponseType"/>
  <xs:element name="Modify" type="dstref:ModifyType"/>
  <xs:element name="ModifyResponse" type="dstref:ModifyResponseType"/>
  <xs:element name="Delete" type="dstref:DeleteType"/>
  <xs:element name="DeleteResponse" type="dstref:DeleteResponseType"/>
<!--endsec(methods)-->
<!--sec(redefs)-->
  <xs:complexType name="SelectType">
    <xs:simpleContent>
      <xs:extension base="xs:string"/>
    </xs:simpleContent>
  </xs:complexType>
  <xs:complexType name="TestOpType">
    <xs:simpleContent>
      <xs:extension base="xs:string"/>
    </xs:simpleContent>
  </xs:complexType>
  <xs:complexType name="SortType">
    <xs:simpleContent>
      <xs:extension base="xs:string"/>
    </xs:simpleContent>
  </xs:complexType>
  <xs:complexType name="AppDataType">
    <xs:simpleContent>
      <xs:extension base="xs:string"/>
    </xs:simpleContent>
  </xs:complexType>
<!--endsec(redefs)-->
<!--sec(create)-->
  <xs:complexType name="CreateType">
    <xs:complexContent>
      <xs:extension base="dst:RequestType">
        <xs:sequence>
          <xs:element ref="dstref:CreateItem" minOccurs="1" maxOccurs="unbounded"/>
          <xs:element ref="dstref:ResultQuery" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:element name="CreateItem" type="dstref:CreateItemType"/>
  <xs:complexType name="CreateItemType">
    <xs:sequence>
      <xs:element ref="dstref:NewData" minOccurs="0" maxOccurs="1"/>
```

```
    </xs:sequence>
    <xs:attributeGroup ref="dst:CreateItemAttributeGroup"/>
  </xs:complexType>
  <xs:element name="NewData" type="dstref:AppDataType"/>
  <xs:complexType name="CreateResponseType">
    <xs:complexContent>
      <xs:extension base="dstref:DataResponseType"/>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="DataResponseType">
    <xs:complexContent>
      <xs:extension base="dst:DataResponseBaseTyp e">
        <xs:sequence>
          <xs:element ref="dstref:ItemData" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
<!--endsec(create)-->
<!--sec(query)-->
  <xs:complexType name="QueryType">
    <xs:complexContent>
      <xs:extension base="dst:RequestType">
        <xs:sequence>
          <xs:element ref="dstref:TestItem" minOccurs="0" maxOccurs="unbounded"/>
          <xs:element ref="dstref:QueryItem" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:element name="TestItem" type="dstref:TestItemType"/>
  <xs:complexType name="TestItemType">
    <xs:complexContent>
      <xs:extension base="dst:TestItemBaseType">
        <xs:sequence>
          <xs:element name="TestOp" minOccurs="0" maxOccurs="1" type="dstref:TestOpType"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:element name="QueryItem" type="dstref:QueryItemType"/>
  <xs:complexType name="QueryItemType">
    <xs:complexContent>
      <xs:extension base="dstref:ResultQueryType">
        <xs:attributeGroup ref="dst:PaginationAttributeGroup"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
<!--endsec(query)-->
<!--sec(queryresp)-->
  <xs:complexType name="QueryResponseType">
    <xs:complexContent>
      <xs:extension base="dst:DataResponseBaseType">
        <xs:sequence>
          <xs:element ref="dst:TestResult" minOccurs="0" maxOccurs="unbounded"/>
          <xs:element ref="dstref:Data" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:element name="Data" type="dstref:DataType"/>
  <xs:complexType name="DataType">
    <xs:complexContent>
      <xs:extension base="dstref:ItemDataType">
        <xs:attributeGroup ref="dst:PaginationResponseAttributeGroup"/>
      </xs:extension>
    </xs:complexContent>
```

```
  </xs:complexType>
<!--endsec(queryresp)-->
<!--sec(mod)-->
  <xs:complexType name="ModifyType">
    <xs:complexContent>
      <xs:extension base="dst:RequestType">
        <xs:sequence>
          <xs:element ref="dstref:ModifyItem" minOccurs="1" maxOccurs="unbounded"/>
          <xs:element ref="dstref:ResultQuery" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:element name="ModifyItem" type="dstref:ModifyItemType"/>
  <xs:complexType name="ModifyItemType">
    <xs:sequence>
      <xs:element ref="dstref:Select" minOccurs="0" maxOccurs="1"/>
      <xs:element ref="dstref:NewData" minOccurs="0" maxOccurs="1"/>
    </xs:sequence>
    <xs:attributeGroup ref="dst:ModifyItemAttributeGroup"/>
  </xs:complexType>
  <xs:complexType name="ModifyResponseType">
    <xs:complexContent>
      <xs:extension base="dstref:DataResponseType"/>
    </xs:complexContent>
  </xs:complexType>
<!--endsec(mod)-->
<!--sec(del)-->
  <xs:complexType name="DeleteType">
    <xs:complexContent>
      <xs:extension base="dst:RequestType">
        <xs:sequence>
          <xs:element ref="dstref:DeleteItem" minOccurs="1" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:element name="DeleteItem" type="dstref:DeleteItemType"/>
  <xs:complexType name="DeleteItemType">
    <xs:complexContent>
      <xs:extension base="dst:DeleteItemBaseType">
        <xs:sequence>
          <xs:element ref="dstref:Select" minOccurs="0" maxOccurs="1"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="DeleteResponseType">
    <xs:complexContent>
      <xs:extension base="lu:ResponseType"/>
    </xs:complexContent>
  </xs:complexType>
<!--endsec(del)-->
<!--sec(resqry)-->
  <xs:element name="Select" type="dstref:SelectType"/>
  <xs:element name="ResultQuery" type="dstref:ResultQueryType"/>
  <xs:complexType name="ResultQueryType">
    <xs:complexContent>
      <xs:extension base="dst:ResultQueryBaseType">
        <xs:sequence>
          <xs:element ref="dstref:Select" minOccurs="0" maxOccurs="1"/>
          <xs:element name="Sort" minOccurs="0" maxOccurs="1" type="dstref:SortType"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:element name="ItemData" type="dstref:ItemDataType"/>
```

```
  <xs:complexType name="ItemDataType">
    <xs:complexContent>
      <xs:extension base="dstref:AppDataType">
        <xs:attributeGroup ref="dst:ItemDataAttributeGroup"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
<!--endsec(resqry)-->
</xs:schema>
```

11.2. DST Utility Schema

The formal utility schema follows.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
   targetNamespace="urn:liberty:dst:2006-08"
   xmlns:dst="urn:liberty:dst:2006-08"
   xmlns:lu="urn:liberty:util:2006-08"
   xmlns:xml="http://www.w3.org/XML/1998/namespace"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   elementFormDefault="qualified"
   attributeFormDefault="unqualified">
  <xs:import namespace="urn:liberty:util:2006-08"
     schemaLocation="liberty-idwsf-utility-v2.0.xsd"/>
  <xs:import namespace="http://www.w3.org/XML/1998/namespace"
     schemaLocation="http://www.w3.org/2001/xml.xsd"/>
<!--sec(ca)-->
  <xs:attribute name="id" type="lu:IDType"/>
  <xs:attribute name="modificationTime" type="xs:dateTime"/>
  <xs:attributeGroup name="commonAttributes">
    <xs:attribute ref="dst:id" use="optional"/>
    <xs:attribute ref="dst:modificationTime" use="optional"/>
  </xs:attributeGroup>
  <xs:attribute name="ACC" type="xs:anyURI"/>
  <xs:attribute name="ACCTime" type="xs:dateTime"/>
  <xs:attribute name="modifier" type="xs:string"/>
  <xs:attributeGroup name="leafAttributes">
    <xs:attributeGroup ref="dst:commonAttributes"/>
    <xs:attribute ref="dst:ACC" use="optional"/>
    <xs:attribute ref="dst:ACCTime" use="optional"/>
    <xs:attribute ref="dst:modifier" use="optional"/>
  </xs:attributeGroup>
  <xs:attribute name="script" type="xs:anyURI"/>
  <xs:attributeGroup name="localizedLeafAttributes">
    <xs:attributeGroup ref="dst:leafAttributes"/>
    <xs:attribute ref="xml:lang" use="required"/>
    <xs:attribute ref="dst:script" use="optional"/>
  </xs:attributeGroup>
  <xs:attribute name="refreshOnOrAfter" type="xs:dateTime"/>
  <xs:attribute name="destroyOnOrAfter" type="xs:dateTime"/>
<!--endsec(ca)-->
<!--sec(ct)-->
  <xs:complexType name="DSTLocalizedString">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attributeGroup ref="dst:localizedLeafAttributes"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
  <xs:complexType name="DSTString">
    <xs:simpleContent>
      <xs:extension base="xs:string">
```

```
      <xs:attributeGroup ref="dst:leafAttributes"/>
    </xs:extension>
  </xs:simpleContent>
 </xs:complexType>
 <xs:complexType name="DSTInteger">
  <xs:simpleContent>
    <xs:extension base="xs:integer">
      <xs:attributeGroup ref="dst:leafAttributes"/>
    </xs:extension>
  </xs:simpleContent>
 </xs:complexType>
 <xs:complexType name="DSTURI">
  <xs:simpleContent>
    <xs:extension base="xs:anyURI">
      <xs:attributeGroup ref="dst:leafAttributes"/>
    </xs:extension>
  </xs:simpleContent>
 </xs:complexType>
 <xs:complexType name="DSTDate">
  <xs:simpleContent>
    <xs:extension base="xs:date">
      <xs:attributeGroup ref="dst:leafAttributes"/>
    </xs:extension>
  </xs:simpleContent>
 </xs:complexType>
 <xs:complexType name="DSTMonthDay">
  <xs:simpleContent>
    <xs:extension base="xs:gMonthDay">
      <xs:attributeGroup ref="dst:leafAttributes"/>
    </xs:extension>
  </xs:simpleContent>
 </xs:complexType>
<!--endsec(ct)-->
<!--sec(msgintf)-->
 <xs:complexType name="RequestType">
  <xs:sequence>
    <xs:element ref="lu:Extension" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute ref="lu:itemID" use="optional"/>
  <xs:anyAttribute namespace="##other" processContents="lax"/>
 </xs:complexType>
 <xs:complexType name="DataResponseBaseType">
  <xs:complexContent>
    <xs:extension base="lu:ResponseType">
      <xs:attribute name="timeStamp" use="optional" type="xs:dateTime"/>
    </xs:extension>
  </xs:complexContent>
 </xs:complexType>
<!--endsec(msgintf)-->
<!--sec(select)-->
 <xs:element name="ChangeFormat">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="ChangedElements"/>
      <xs:enumeration value="CurrentElements"/>
    </xs:restriction>
  </xs:simpleType>
 </xs:element>
 <xs:attribute name="changeFormat">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="ChangedElements"/>
      <xs:enumeration value="CurrentElements"/>
      <xs:enumeration value="All"/>
    </xs:restriction>
  </xs:simpleType>
 </xs:attribute>
```

```
  <xs:attribute name="objectType" type="xs:NCName"/>
  <xs:attribute name="predefined" type="xs:string"/>
  <xs:attributeGroup name="selectQualif">
    <xs:attribute ref="dst:objectType" use="optional"/>
    <xs:attribute ref="dst:predefined" use="optional"/>
  </xs:attributeGroup>
<!--endsec(select)-->
<!--sec(resquery)-->
  <xs:complexType name="ResultQueryBaseType">
    <xs:sequence>
      <xs:element ref="dst:ChangeFormat" minOccurs="0" maxOccurs="2"/>
    </xs:sequence>
    <xs:attributeGroup ref="dst:selectQualif"/>
    <xs:attribute ref="lu:itemIDRef" use="optional"/>
    <xs:attribute name="contingency" use="optional" type="xs:boolean"/>
    <xs:attribute name="includeCommonAttributes" use="optional" type="xs:boolean" default="0"/>
    <xs:attribute name="changedSince" use="optional" type="xs:dateTime"/>
    <xs:attribute ref="lu:itemID" use="optional"/>
  </xs:complexType>
  <xs:attributeGroup name="ItemDataAttributeGroup">
    <xs:attribute ref="lu:itemIDRef" use="optional"/>
    <xs:attribute name="notSorted" use="optional">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value="Now"/>
          <xs:enumeration value="Never"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute ref="dst:changeFormat" use="optional"/>
  </xs:attributeGroup>
<!--endsec(resquery)-->
<!--sec(testitem)-->
  <xs:complexType name="TestItemBaseType">
    <xs:attributeGroup ref="dst:selectQualif"/>
    <xs:attribute name="id" use="optional" type="xs:ID"/>
    <xs:attribute ref="lu:itemID" use="optional"/>
  </xs:complexType>
  <xs:element name="TestResult" type="dst:TestResultType"/>
  <xs:complexType name="TestResultType">
    <xs:simpleContent>
      <xs:extension base="xs:boolean">
        <xs:attribute ref="lu:itemIDRef" use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
<!--endsec(testitem)-->
<!--sec(pagination)-->
  <xs:attributeGroup name="PaginationAttributeGroup">
    <xs:attribute name="count" use="optional" type="xs:nonNegativeInteger"/>
    <xs:attribute name="offset" use="optional" type="xs:nonNegativeInteger" default="0"/>
    <xs:attribute name="setID" use="optional" type="lu:IDType"/>
    <xs:attribute name="setReq" use="optional">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value="Static"/>
          <xs:enumeration value="DeleteSet"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
  </xs:attributeGroup>
  <xs:attributeGroup name="PaginationResponseAttributeGroup">
    <xs:attribute name="remaining" use="optional" type="xs:integer"/>
    <xs:attribute name="nextOffset" use="optional" type="xs:nonNegativeInteger" default="0"/>
    <xs:attribute name="setID" use="optional" type="lu:IDType"/>
  </xs:attributeGroup>
<!--endsec(pagination)-->
```

```
<!--sec(create)-->
  <xs:attributeGroup name="CreateItemAttributeGroup">
    <xs:attribute ref="dst:objectType" use="optional"/>
    <xs:attribute name="id" use="optional" type="xs:ID"/>
    <xs:attribute ref="lu:itemID" use="optional"/>
  </xs:attributeGroup>
<!--endsec(create)-->
<!--sec(mod)-->
  <xs:attributeGroup name="ModifyItemAttributeGroup">
    <xs:attributeGroup ref="dst:selectQualif"/>
    <xs:attribute name="notChangedSince" use="optional" type="xs:dateTime"/>
    <xs:attribute name="overrideAllowed" use="optional" type="xs:boolean" default="0"/>
    <xs:attribute name="id" use="optional" type="xs:ID"/>
    <xs:attribute ref="lu:itemID" use="optional"/>
  </xs:attributeGroup>
<!--endsec(mod)-->
<!--sec(del)-->
  <xs:complexType name="DeleteItemBaseType">
    <xs:attributeGroup ref="dst:selectQualif"/>
    <xs:attribute name="notChangedSince" use="optional" type="xs:dateTime"/>
    <xs:attribute name="id" use="optional" type="xs:ID"/>
    <xs:attribute ref="lu:itemID" use="optional"/>
  </xs:complexType>
  <xs:complexType name="DeleteResponseType">
    <xs:complexContent>
      <xs:extension base="lu:ResponseType"/>
    </xs:complexContent>
  </xs:complexType>
<!--endsec(del)-->
</xs:schema>
```

References

Normative

[LibertyDisco] Hodges, Jeff, Cahill, Conor, eds. "Liberty ID-WSF Discovery Service Specification," Version 2.0, Liberty Alliance Project (30 July, 2006). *http://www.projectliberty.org/specs*

[LibertySOAPBinding] Hodges, Jeff, Kemp, John, Aarts, Robert, Whitehead, Greg, Madsen, Paul, eds. "Liberty ID-WSF SOAP Binding Specification," Version 2.0, Liberty Alliance Project (30 July, 2006). *http://www.projectliberty.org/specs*

[LibertyPAOS] Aarts, Robert, Kemp, John, eds. "Liberty Reverse HTTP Binding for SOAP Specification," Version 2.0, Liberty Alliance Project (30 July, 2006). *http://www.projectliberty.org/specs*

[LibertyInteract] Aarts, Robert, Madsen, Paul, eds. "Liberty ID-WSF Interaction Service Specification," Version 2.0, Liberty Alliance Project (30 July, 2006). *http://www.projectliberty.org/specs*

[LibertySecMech] Hirsch, Frederick, eds. "Liberty ID-WSF Security Mechanisms Core," Version v2.0, Liberty Alliance Project (30 July, 2006). *http://www.projectliberty.org/specs*

[LibertyGlossary] Hodges, Jeff, eds. "Liberty Technical Glossary," Version v2.0, Liberty Alliance Project (30 July, 2006). *http://www.projectliberty.org/specs*

[LibertyReg] Kemp, John, eds. "Liberty Enumeration Registry Governance," Version 1.1, Liberty Alliance Project (14 December, 2004). *http://www.projectliberty.org/specs*

[Schema1-2] Thompson, Henry S., Beech, David, Maloney, Murray, Mendelsohn, Noah, eds. (28 October 2004). "XML Schema Part 1: Structures Second Edition," Recommendation, World Wide Web Consortium *http://www.w3.org/TR/xmlschema-1/*

[RFC2119] S. Bradner "Key words for use in RFCs to Indicate Requirement Levels," RFC 2119, The Internet Engineering Task Force (March 1997). *http://www.ietf.org/rfc/rfc2119.txt*

[XML] Bray, Tim, Paoli, Jean, Sperberg-McQueen, C. M., Maler, Eve, Yergeau, Francois, eds. (04 February 2004). "Extensible Markup Language (XML) 1.0 (Third Edition)," Recommendation, World Wide Web Consortium *http://www.w3.org/TR/2004/REC-xml-20040204*