



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

1 **Notice**

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

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

15 Copyright © 2006 Adobe Systems; America Online, Inc.; American Express Company; Amsoft Systems Pvt Ltd.;
16 Avatier Corporation; Axalto; Bank of America Corporation; BIPAC; BMC Software, Inc.; Computer Associates
17 International, Inc.; DataPower Technology, Inc.; Diversinet Corp.; Enosis Group LLC; Entrust, Inc.; Epok, Inc.;
18 Ericsson; Fidelity Investments; Forum Systems, Inc.; France Télécom; French Government Agence pour le
19 développement de l'administration électronique (ADAE); Gamefederation; Gemplus; General Motors; Giesecke &
20 Devrient GmbH; GSA Office of Governmentwide Policy; Hewlett-Packard Company; IBM Corporation; Intel
21 Corporation; Intuit Inc.; Kantega; Kayak Interactive; MasterCard International; Mobile Telephone Networks (Pty)
22 Ltd; NEC Corporation; Netegrity, Inc.; NeuStar, Inc.; Nippon Telegraph and Telephone Corporation; Nokia
23 Corporation; Novell, Inc.; NTT DoCoMo, Inc.; OpenNetwork; Oracle Corporation; Ping Identity Corporation;
24 Reactivity Inc.; Royal Mail Group plc; RSA Security Inc.; SAP AG; Senforce; Sharp Laboratories of America;
25 Sigaba; SmartTrust; Sony Corporation; Sun Microsystems, Inc.; Supremacy Financial Corporation; Symlabs, Inc.;
26 Telecom Italia S.p.A.; Telefónica Móviles, S.A.; Trusted Network Technologies; UTI; VeriSign, Inc.; Vodafone
27 Group Plc.; Wave Systems Corp. All rights reserved.

28 Liberty Alliance Project
29 Licensing Administrator
30 c/o IEEE-ISTO
31 445 Hoes Lane
32 Piscataway, NJ 08855-1331, USA
33 info@projectliberty.org

34 Contents

35	1. Introduction	5
36	1.1. Notation	5
37	1.2. Liberty Considerations	5
38	1.3. Namespaces	5
39	1.4. Applying DST to Define Services	6
40	1.5. Applying the DST Reference Model	6
41	2. Data Model	8
42	2.1. Guidelines for Schemata	8
43	2.2. Extending a Service	8
44	2.3. Time Values and Synchronization	9
45	2.4. Common XML Attributes	9
46	2.4.1. The <code>commonAttributes</code> XML Attribute Group	10
47	2.4.2. The <code>leafAttributes</code> XML Attribute Group	10
48	2.4.3. The <code>localizedLeafAttributes</code> XML Attribute Group	11
49	2.4.4. Individual Common XML attributes	12
50	2.5. Common Data Types	12
51	3. Message Interface	14
52	3.1. Multiple Occurrences of Request or Response	14
53	3.2. Status and Fault Reporting	14
54	3.2.1. Top Level <code><Status></code> Element	15
55	3.2.2. Second Level <code><Status></code> Codes	16
56	3.3. The <code>timeStamp</code> XML Attribute	17
57	3.4. General Error Handling	18
58	3.5. Linking with <code>ids</code>	18
59	3.6. Resources	18
60	3.7. Selection	19
61	3.8. Common Processing Rules for Selection	20
62	3.8.1. Processing Rules for the <code>predefined</code> XML Attribute	20
63	3.8.2. Processing Rules for the <code>objectType</code> XML Attribute	21
64	3.8.3. Processing Rules for the <code><Select></code> Element	21
65	3.9. Requesting Meta and Additional Data	21
66	3.10. Common Processing Rules for Requesting Meta and Additional Data	22
67	4. Querying Data	24
68	4.1. The <code><Query></code> Element	24
69	4.1.1. The <code><TestItem></code> Element	25
70	4.1.2. The <code><QueryItem></code> Element	26
71	4.1.3. Pagination	27
72	4.2. The <code><QueryResponse></code> Element	28
73	4.3. <code><ResultQuery></code> or <code><QueryItem></code> Conditioned by <code><TestItem></code>	28
74	4.4. Processing Rules for Queries	29
75	4.4.1. Processing Rules for Multiple <code><QueryItem></code> Elements	29
76	4.4.2. Processing Rules for <code><Select></code> Element	29
77	4.4.3. Sorting Query Results	30
78	4.4.4. Pagination of Query Results	30
79	4.4.5. Effect of Access and Privacy Policies	32
80	4.4.6. Querying Changes Since Specified Time	32
81	4.4.7. Requesting Common XML Attributes	34
82	4.5. Examples	35
83	5. Creating Data Objects	42
84	5.1. <code><Create></code> Element	42
85	5.2. <code><CreateResponse></code> Element	42
86	5.3. Processing Rules for Creating Data Objects	43

87	5.3.1. Multiple <CreateItem> Elements	43
88	5.3.2. Only One Type of Data Object per <CreateItem>	43
89	5.3.3. Handling commonAttributes and leafAttributes upon Creation	43
90	5.3.4. WSC Might Not Be Allowed to Add Certain Data or Any Data	44
91	5.3.5. WSP May Place Some Restrictions for the data It Is Hosting	44
92	6. Deleting Data Objects	46
93	6.1. <Delete> Element	46
94	6.2. <DeleteResponse> Element	46
95	6.3. Processing Rules for Deletion	46
96	6.3.1. Supporting Multiple <DeleteItem> Elements	46
97	6.3.2. Only One Type of Data Object May Be Deleted with One <DeleteItem>	47
98	6.3.3. Avoiding Deletion of Data if It Has Changed In-between	47
99	6.3.4. WSC Might Not Be Allowed to Delete Certain or Any Data	47
100	7. Modifying Data	49
101	7.1. <Modify> Element	49
102	7.2. <ModifyResponse> Element	50
103	7.3. Processing Rules for Modifications	50
104	7.3.1. Multiple <ModifyItem> Elements	50
105	7.3.2. What Exactly Is Modified	51
106	7.3.3. Handling commonAttributes and leafAttributes in Modify	52
107	7.3.4. Accounting for Concurrent Updates	52
108	7.3.5. WSC Might Not Be Allowed to Make Only Certain or Any Modifications	53
109	7.3.6. WSP May Impose Some Restrictions for the Data It Is Hosting	53
110	7.4. Examples of Modifications	53
111	8. WSF-1.1 Compatibility	56
112	9. Actions	57
113	10. Checklist for Service Specifications	58
114	11. Schemata	61
115	11.1. DST Reference Model Schema	61
116	11.2. DST Utility Schema	64
117	References	68

118 **1. Introduction**

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

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

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

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

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

139 **1.1. Notation**

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

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

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

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

148 **1.2. Liberty Considerations**

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

153 **1.3. Namespaces**

154 The namespaces described in table 1 are used.

155

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	Liberty Utility schema

156 1.4. Applying DST to Define Services

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

159 a. Answer every question specified in the check list, see [Section 10](#)

160 b. Waive inappropriate processing rules

161 c. Define additional processing rules

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

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

166 1.5. Applying the DST Reference Model

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

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

```

175 <xs:element name="Create" type="dstref:CreateType" />
176 <xs:element name="CreateResponse" type="dstref:CreateResponseType" />
177 <xs:element name="Query" type="dstref:QueryType" />
178 <xs:element name="QueryResponse" type="dstref:QueryResponseType" />
179 <xs:element name="Modify" type="dstref:ModifyType" />
180 <xs:element name="ModifyResponse" type="dstref:ModifyResponseType" />
181 <xs:element name="Delete" type="dstref>DeleteType" />
182 <xs:element name="DeleteResponse" type="dstref>DeleteResponseType" />

```

183

Figure 1. Reference Definitions of Methods

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

```
186 <xs:complexType name="SelectType">  
187   <xs:simpleContent>  
188     <xs:extension base="xs:string"/>  
189   </xs:simpleContent>  
190 </xs:complexType>  
191 <xs:complexType name="TestOpType">  
192   <xs:simpleContent>  
193     <xs:extension base="xs:string"/>  
194   </xs:simpleContent>  
195 </xs:complexType>  
196 <xs:complexType name="SortType">  
197   <xs:simpleContent>  
198     <xs:extension base="xs:string"/>  
199   </xs:simpleContent>  
200 </xs:complexType>  
201 <xs:complexType name="AppDataType">  
202   <xs:simpleContent>  
203     <xs:extension base="xs:string"/>  
204   </xs:simpleContent>  
205 </xs:complexType>  
206
```

207

Figure 2. DST Parameterization Points

208 **2. Data Model**

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

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

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

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

232 **2.1. Guidelines for Schemata**

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

- 235 1. Each data attribute regarding the Principal **SHOULD** be defined as an XML element of a suitable type.
- 236 2. XML attributes **SHOULD** be used only to qualify the data attribute defined as XML elements and not contain the
237 actual data values related to the Principal.
- 238 3. An XML element **SHOULD** either contain other XML elements or actual data value. An XML element **SHOULD**
239 **NOT** have mixed content, i.e., both a value and sub-elements. Also complex types **all** and **choice** **SHOULD**
240 **NOT** be used.
- 241 4. Once a data attribute has been published in a specification for a service type, its syntax and semantics **MUST** not
242 change. If evolution in syntax or semantics is needed, any new version of a data attribute **MUST** be assigned a
243 different name, effectively creating a new attribute with new semantics so that it does not conflict with the original
244 attribute definition.
- 245 5. All elements **MUST** be defined as global elements, when they can be requested individually. When elements with
246 complex type are defined, references to global elements are used. The reason for this guideline is that the XML
247 Schema for a service does not only define the syntax of the data supported by the service but also the transfer
248 syntax. In many cases it should be possible to query and modify individual elements.
- 249 6. The type definitions provided by the XML Schema **SHOULD** be used, when they cover the requirements.

250 **2.2. Extending a Service**

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

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

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

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

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

270 **The <Extension> Element**

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

274 **2.3. Time Values and Synchronization**

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

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

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

285 **2.4. Common XML Attributes**

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

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

```
300 <xs:attribute name="id" type="lu:IDType"/>
301 <xs:attribute name="modificationTime" type="xs:dateTime"/>
302 <xs:attributeGroup name="commonAttributes">
303   <xs:attribute ref="dst:id" use="optional"/>
304   <xs:attribute ref="dst:modificationTime" use="optional"/>
305 </xs:attributeGroup>
306 <xs:attribute name="ACC" type="xs:anyURI"/>
307 <xs:attribute name="ACCTime" type="xs:dateTime"/>
308 <xs:attribute name="modifier" type="xs:string"/>
309 <xs:attributeGroup name="leafAttributes">
310   <xs:attributeGroup ref="dst:commonAttributes"/>
311   <xs:attribute ref="dst:ACC" use="optional"/>
312   <xs:attribute ref="dst:ACCTime" use="optional"/>
313   <xs:attribute ref="dst:modifier" use="optional"/>
314 </xs:attributeGroup>
315 <xs:attribute name="script" type="xs:anyURI"/>
316 <xs:attributeGroup name="localizedLeafAttributes">
317   <xs:attributeGroup ref="dst:leafAttributes"/>
318   <xs:attribute ref="xml:lang" use="required"/>
319   <xs:attribute ref="dst:script" use="optional"/>
320 </xs:attributeGroup>
321 <xs:attribute name="refreshOnOrAfter" type="xs:dateTime"/>
322 <xs:attribute name="destroyOnOrAfter" type="xs:dateTime"/>
```

323 **Figure 3. DST Common XML Attributes**

324 2.4.1. The `commonAttributes` XML Attribute Group

325 There are only two common XML attributes:

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

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

344 2.4.2. The `leafAttributes` XML Attribute Group

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

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

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

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

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

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

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

365 `urn:liberty:dst:acc:secondarydocuments` The value has been validated from secondary docu-
366 ments (such as the address from an electric bill).

367 `urn:liberty:dst:acc:primarydocuments` The value has been validated from primary docu-
368 ments (for example, the name and identification number from a
369 passport).

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

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

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

378 2.4.3. The `localizedLeafAttributes` XML Attribute Group

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

- 384 `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
385 attribute.
386
- 387 `script` (optional) Sometimes the language does not define the writing system used. In such cases, this XML
388 attribute defines the writing system in more detail. This specification defines the following
389 values for this XML attribute: `urn:liberty:dst:script:kana` and `urn:liberty:dst:script:kanji`.
390 See [[LibertyReg](#)] where to find additional values, if any, and how to specify more values.

391 **2.4.4. Individual Common XML attributes**

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

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

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

407 **2.5. Common Data Types**

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

```
413 <xs:complexType name="DSTLocalizedString">
414   <xs:simpleContent>
415     <xs:extension base="xs:string">
416       <xs:attributeGroup ref="dst:localizedLeafAttributes"/>
417     </xs:extension>
418   </xs:simpleContent>
419 </xs:complexType>
420 <xs:complexType name="DSTString">
421   <xs:simpleContent>
422     <xs:extension base="xs:string">
423       <xs:attributeGroup ref="dst:leafAttributes"/>
424     </xs:extension>
425   </xs:simpleContent>
426 </xs:complexType>
427 <xs:complexType name="DSTInteger">
428   <xs:simpleContent>
429     <xs:extension base="xs:integer">
430       <xs:attributeGroup ref="dst:leafAttributes"/>
431     </xs:extension>
432   </xs:simpleContent>
433 </xs:complexType>
434 <xs:complexType name="DSTURI">
435   <xs:simpleContent>
436     <xs:extension base="xs:anyURI">
437       <xs:attributeGroup ref="dst:leafAttributes"/>
438     </xs:extension>
439   </xs:simpleContent>
440 </xs:complexType>
441 <xs:complexType name="DSTDate">
442   <xs:simpleContent>
443     <xs:extension base="xs:date">
444       <xs:attributeGroup ref="dst:leafAttributes"/>
445     </xs:extension>
446   </xs:simpleContent>
447 </xs:complexType>
448 <xs:complexType name="DSTMonthDay">
449   <xs:simpleContent>
450     <xs:extension base="xs:gMonthDay">
451       <xs:attributeGroup ref="dst:leafAttributes"/>
452     </xs:extension>
453   </xs:simpleContent>
454 </xs:complexType>
```

455

Figure 4. General Data Types with DST Attributes

456 3. Message Interface

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

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

464 **Table 2. Requests and Responses**

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

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

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

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

```

476 <xs:complexType name="RequestType">
477   <xs:sequence>
478     <xs:element ref="lu:Extension" minOccurs="0" maxOccurs="unbounded" />
479   </xs:sequence>
480   <xs:attribute ref="lu:itemID" use="optional" />
481   <xs:anyAttribute namespace="##other" processContents="lax" />
482 </xs:complexType>
483 <xs:complexType name="DataResponseBaseType">
484   <xs:complexContent>
485     <xs:extension base="lu:ResponseType">
486       <xs:attribute name="timeStamp" use="optional" type="xs:dateTime" />
487     </xs:extension>
488   </xs:complexContent>
489 </xs:complexType>
490

```

491 **Figure 5. Commonality of Requests and Responses**

492 3.1. Multiple Occurrences of Request or Response

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

496 3.2. Status and Fault Reporting

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

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

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

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

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

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

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

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

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

525 3.2.1. Top Level <Status> Element

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

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

533	OK	The value OK means that the processing of a <i>RequestElement</i> has succeeded. A second level status code MAY be used to indicate some special cases, but the processing of a <i>RequestElement</i> has succeeded.
534		
535		
536	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 <i>RequestElement</i> . 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 <i>RequestElement</i> .
537		
538		
539		
540		
541		
542		
543		
544		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.
545		
546		
547		
548		A WSP MAY also choose to fail completely another type of <i>RequestElement</i> , 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.
549		
550		
551		
552		
553		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.
554		
555		
556		
557	Failed	The value Failed means that the processing of a <i>RequestElement</i> has failed. Either the processing of the whole <i>RequestElement</i> 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.
558		
559		
560		
561		

562 3.2.2. Second Level <Status> Codes

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

- 564 ActionNotAuthorized
- 565 AggregationNotSupported
- 566 AllReturned
- 567 ChangeHistoryNotSupported
- 568 ChangedSinceReturnsAll
- 569 DataTooLong
- 570 DoesNotExist
- 571 EmptyRequest
- 572 ExistsAlready
- 573 ExtensionNotSupported
- 574 Failed
- 575 FormatNotSupported
- 576 InvalidData
- 577 InvalidExpires
- 578 InvalidItemIDRef
- 579 InvalidObjectType
- 580 InvalidPredefined
- 581 InvalidSelect
- 582 InvalidSetID
- 583 InvalidSetReq
- 584 InvalidSort
- 585 ItemIDDuplicated

586 ResultQueryNotSupported
587 MissingCredentials
588 MissingDataElement
589 MissingExpiration
590 MissingItemID
591 MissingNewDataElement
592 MissingObjectType
593 MissingSecurityMechIDElement
594 MissingSelect
595 ModifiedSince
596 NewOrExisting
597 NoMoreElements
598 NoMoreObjects
599 NoMultipleAllowed
600 NoMultipleResources
601 NoSuchTest
602 ObjectTypeMismatch
603 OK
604 PaginationNotSupported
605 Partial
606 RequestedAggregationNotSupported
607 RequestedPaginationNotSupported
608 RequestedSortingNotSupported
609 RequestedTriggerNotSupported
610 SecurityMechIDNotAccepted
611 SetOrNewQuery
612 SortNotSupported
613 StaticNotSupported
614 TimeOut
615 TriggerNotSupported
616 UnexpectedError
617 UnspecifiedError
618 UnsupportedObjectType
619 UnsupportedPredefined
620
621

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

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

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

634 **3.3. The `timeStamp` XML Attribute**

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

639 **The processing rule for the `timeStamp` XML Attribute**

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

648 3.4. General Error Handling

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

- 652 • `urn:liberty:dst:noQuery`
- 653 • `urn:liberty:dst:noCreate`
- 654 • `urn:liberty:dst:noDelete`
- 655 • `urn:liberty:dst:noModify`

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

- 657 1. If the processing takes too long (for example some back-end system is not responding fast enough) the second
658 level status code `Timeout` SHOULD be used to indicate this, when the request is not processed due to a
659 WSP internal time out. The duration and indeed criteria for deciding when timeout has happened depend on
660 WSP and are not externally visible other than the fact that the `Timeout` status code is returned. Note that
661 [[LibertySOAPBinding](#)] specifies a header block which a WSC may use to define threshold for timeout, but that
662 is different functionality and the processing rules for that are specified in [[LibertySOAPBinding](#)].
- 663 2. Other error conditions than those listed in this specification and in service specifications may occur. There are
664 two status codes defined for those cases. For cases a WSP (or WSC receiving a notification) can handle normally
665 but for which there is no status code specified, the second level status code `UnspecifiedError` SHOULD be
666 used. For totally unexpected cases the second level status code `UnexpectedError` SHOULD be used.

667 3.5. Linking with `ids`

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

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

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

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

681 3.6. Resources

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

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

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

693 3.7. Selection

694 The second level of the selection is deeper inside the *RequestElement* element. The request message must describe in
695 more detail what it wants to access inside the specified resource. This can be specified in two different ways. Either
696 the requesting WSC accesses data by selecting it explicitly in the request or uses *predefined* selection. When the
697 predefined selections are supported, the available predefined selections are specified in the service specification or are
698 agreed out of band. A WSC specifies the predefined selection it wants to use by putting its identifier into the request.
699 The identifier is carried as the value of the predefined XML attribute. When a WSC explicitly selects the data, it
700 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.
701 The XML attribute *objectType* and the element <Select> are specified for making the explicit selection.

```
702 <xs:element name="ChangeFormat">
703   <xs:simpleType>
704     <xs:restriction base="xs:string">
705       <xs:enumeration value="ChangedElements"/>
706       <xs:enumeration value="CurrentElements"/>
707     </xs:restriction>
708   </xs:simpleType>
709 </xs:element>
710 <xs:attribute name="changeFormat">
711   <xs:simpleType>
712     <xs:restriction base="xs:string">
713       <xs:enumeration value="ChangedElements"/>
714       <xs:enumeration value="CurrentElements"/>
715       <xs:enumeration value="All"/>
716     </xs:restriction>
717   </xs:simpleType>
718 </xs:attribute>
719 <xs:attribute name="objectType" type="xs:NCName"/>
720 <xs:attribute name="predefined" type="xs:string"/>
721 <xs:attributeGroup name="selectQualif">
722   <xs:attribute ref="dst:objectType" use="optional"/>
723   <xs:attribute ref="dst:predefined" use="optional"/>
724 </xs:attributeGroup>
```

725 **Figure 6. XML Attributes for <Select>**

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

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

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

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

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

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

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

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

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

763 **3.8. Common Processing Rules for Selection**

764 **3.8.1. Processing Rules for the predefined XML Attribute**

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

- 769 2. If the predefined XML attribute contains an identifier of a predefined selection, which a WSP does not
770 support, the processing of the subelement containing the predefined XML attribute MUST fail and a status
771 code indicating the failure MUST be returned in the response. A more detailed status code with the value
772 `UnsupportedPredefined` SHOULD be used in addition to the top level status code. If the predefined
773 XML attribute contains an unknown value, the processing of the subelement containing the predefined XML
774 attribute MUST fail and a status code indicating failure MUST be returned in the response. A more detailed status
775 code with the value `InvalidPredefined` SHOULD be used in addition to the top level status code.
- 776 3. A WSP MUST follow service specific processing rules for the values of the predefined XML attribute.

777 3.8.2. Processing Rules for the `objectType` XML Attribute

- 778 1. If the `objectType` XML attribute is missing from a subelement of a *RequestElement* element, when it is
779 supposed to be used, the processing of that subelement MUST fail and a status code indicating the failure MUST
780 be returned in the response. A more detailed status code with the value `MissingObjectType` SHOULD be used
781 in addition to the top level status code. The subelements referred here are the `<QueryItem>`, the `<CreateItem>`,
782 the `<DeleteItem>`, the `<ModifyItem>`, and the `<ResultQuery>`. All these elements are defined later with other
783 protocol elements. Note: in some cases the `objectType` XML attribute is not needed, e.g., when a default object
784 type has been defined for a service and that object type is accessed or a service only supports one `objectType`.
- 785 2. If the `objectType` XML attribute refers to a specified object type, but the WSP does not support it, the
786 processing of the subelement containing the `objectType` XML attribute MUST fail. A more detailed status
787 code with the value `UnsupportedObjectType` SHOULD be used in addition to the top level status code. If the
788 `objectType` XML attribute contains an unknown object type name, the processing of the subelement containing
789 the `objectType` XML attribute MUST fail. A more detailed status code with the value `InvalidObjectType`
790 SHOULD be used in addition to the top level status code. Note that a data service may support extensions,
791 making it difficult for a requestor to know the exact set of allowable values for the `objectType` XML attribute.

792 3.8.3. Processing Rules for the `<Select>` Element

- 793 1. If the `<Select>` element is missing from a subelement of a *RequestElement* element, when it is supposed to be use,
794 the processing of that subelement MUST fail and a status code indicating the failure MUST be returned in the
795 response. A more detailed status code with the value `MissingSelect` SHOULD be used in addition to the top
796 level status code. The subelements referred here are the `<DeleteItem>`, the `<QueryItem>`, the `<ResultQuery>`,
797 and the `<ModifyItem>`. All these elements are defined later with other protocol elements. Note: in some cases
798 the `<Select>` element is not needed.
- 799 2. If the `<Select>` element has invalid content, e.g., does not match with the object type specified by the `objectType`
800 XML attribute, contains an invalid pointer to a data not supported by the WSP or doesn't contain the specified
801 parameters, the processing of the subelement containing the `<Select>` element MUST fail and a status code
802 indicating failure MUST be returned in the response. A more detailed status code with the value `InvalidSelect`
803 SHOULD be used in addition to the top level status code, unless a service specification specifies more detailed
804 status codes better suited for the case. Note that a data service may support extensions, making it difficult for a
805 requestor to know the exact set of allowable values for the `<Select>` element.

806 3.9. Requesting Meta and Additional Data

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

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

814 rules are introduced in [Section 4](#). The data queried with one `<ResultQuery>` element is returned in one `<ItemData>`
 815 element.

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

```

820 <xs:complexType name="ResultQueryBaseType">
821   <xs:sequence>
822     <xs:element ref="dst:ChangeFormat" minOccurs="0" maxOccurs="2"/>
823   </xs:sequence>
824   <xs:attributeGroup ref="dst:selectQualif"/>
825   <xs:attribute ref="lu:itemIDRef" use="optional"/>
826   <xs:attribute name="contingency" use="optional" type="xs:boolean"/>
827   <xs:attribute name="includeCommonAttributes" use="optional" type="xs:boolean" default="0"/>
828   <xs:attribute name="changedSince" use="optional" type="xs:dateTime"/>
829   <xs:attribute ref="lu:itemID" use="optional"/>
830 </xs:complexType>
831 <xs:attributeGroup name="ItemDataAttributeGroup">
832   <xs:attribute ref="lu:itemIDRef" use="optional"/>
833   <xs:attribute name="notSorted" use="optional">
834     <xs:simpleType>
835       <xs:restriction base="xs:string">
836         <xs:enumeration value="Now"/>
837         <xs:enumeration value="Never"/>
838       </xs:restriction>
839     </xs:simpleType>
840   </xs:attribute>
841   <xs:attribute ref="dst:changeFormat" use="optional"/>
842 </xs:attributeGroup>

```

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

```

844 <xs:element name="Select" type="dstref:SelectType"/>
845 <xs:element name="ResultQuery" type="dstref:ResultQueryType"/>
846 <xs:complexType name="ResultQueryType">
847   <xs:complexContent>
848     <xs:extension base="dst:ResultQueryBaseType">
849       <xs:sequence>
850         <xs:element ref="dstref:Select" minOccurs="0" maxOccurs="1"/>
851         <xs:element name="Sort" minOccurs="0" maxOccurs="1" type="dstref:SortType"/>
852       </xs:sequence>
853     </xs:extension>
854   </xs:complexContent>
855 </xs:complexType>
856 <xs:element name="ItemData" type="dstref:ItemDataType"/>
857 <xs:complexType name="ItemDataType">
858   <xs:complexContent>
859     <xs:extension base="dstref:AppDataType">
860       <xs:attributeGroup ref="dst:ItemDataAttributeGroup"/>
861     </xs:extension>
862   </xs:complexContent>
863 </xs:complexType>

```

864 **Figure 8. Reference Model ResultQuery and ItemData**

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

867 **3.10. Common Processing Rules for Requesting Meta and Additional** 868 **Data**

-
- 869 1. A **<ResultQuery>** element MUST be processed as if it was a **<QueryItem>** element and the **<Data>** element
870 used to carry the responses is replaced with **<ItemData>** taking into account the facts that failing **<ResultQuery>**
871 elements do not usually cause a failure of the request message and that **<ResultQuery>** and **<ItemData>** have
872 less features. See [Section 4](#) for details.
- 873 2. If the processing of an **<ResultQuery>** element fails, the rest of the request message MUST be processed
874 normally unless otherwise specified in the service specification. Proper second level status codes SHOULD
875 be used indicate The reason for failing to process the **<ResultQuery>** element, but this MUST NOT affect the
876 value of the top level status code unless otherwise specified in the service specification.
- 877 3. If a WSP does not support **<ResultQuery>** inside **<Create>** or **<Modify>** elements and it receives such, it MUST
878 ignore it and process the message otherwise normally. Not responding to an **<ResultQuery>** is not considered
879 failure and MUST NOT affect the value of the top level status code unless otherwise specified in the service
880 specification. The second level status code `ResultQueryNotSupported` MUST be used to indicate that the
881 WSP does not support this feature, if the feature is allowed in the service specification.
- 882 4. Each **<ResultQuery>** element MUST have the `itemID` XML attribute. Each **<ItemData>** element MUST have
883 an `itemIDRef` XML attribute referring to the corresponding **<ResultQuery>** in the request.
- 884 5. A WSP MAY return additional data in a **<CreateResponse>** and a **<ModifyResponse>** without a WSC
885 requesting for it. A WSC MUST tolerate such unsolicited **<ItemData>** even if it does not interpret it. Unsolicited
886 **<ItemData>** MUST NOT have an `itemIDRef` XML attribute.
- 887 Unsolicited data can be useful, if the WSP thinks that the WSC needs this data, e.g., to be able access the same
888 data later on. For example a WSP may assign locally unique `id` to a newly created object and it wants to return
889 it to the WSC so that the WSC could access the same object easily later on
- 890 6. If **<ResultQuery>** is used inside **<Create>** or **<Modify>** and it uses relative query expressions, the query MUST
891 be interpreted relative to the data object just created or modified.
- 892 7. If **<ResultQuery>** is used inside **<Create>** or **<Modify>**, the `objectType` XML attribute of former MUST
893 agree with the one in the latter.

894 4. Querying Data

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

900 4.1. The <Query> Element

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

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

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

```
910 <xs:complexType name="TestItemBaseType">  
911   <xs:attributeGroup ref="dst:selectQualif"/>  
912   <xs:attribute name="id" use="optional" type="xs:ID"/>  
913   <xs:attribute ref="lu:itemID" use="optional"/>  
914 </xs:complexType>  
915 <xs:element name="TestResult" type="dst:TestResultType"/>  
916 <xs:complexType name="TestResultType">  
917   <xs:simpleContent>  
918     <xs:extension base="xs:boolean">  
919       <xs:attribute ref="lu:itemIDRef" use="required"/>  
920     </xs:extension>  
921   </xs:simpleContent>  
922 </xs:complexType>
```

923 **Figure 9. Utility Schema for TestItem and TestResult**

```

924 <xs:complexType name="QueryType">
925   <xs:complexContent>
926     <xs:extension base="dst:RequestType">
927       <xs:sequence>
928         <xs:element ref="dstref:TestItem" minOccurs="0" maxOccurs="unbounded" />
929         <xs:element ref="dstref:QueryItem" minOccurs="0" maxOccurs="unbounded" />
930       </xs:sequence>
931     </xs:extension>
932   </xs:complexContent>
933 </xs:complexType>
934 <xs:element name="TestItem" type="dstref:TestItemType" />
935 <xs:complexType name="TestItemType">
936   <xs:complexContent>
937     <xs:extension base="dst:TestItemBaseType">
938       <xs:sequence>
939         <xs:element name="TestOp" minOccurs="0" maxOccurs="1" type="dstref:TestOpType" />
940       </xs:sequence>
941     </xs:extension>
942   </xs:complexContent>
943 </xs:complexType>
944 <xs:element name="QueryItem" type="dstref:QueryItemType" />
945 <xs:complexType name="QueryItemType">
946   <xs:complexContent>
947     <xs:extension base="dstref:ResultQueryType">
948       <xs:attributeGroup ref="dst:PaginationAttributeGroup" />
949     </xs:extension>
950   </xs:complexContent>
951 </xs:complexType>

```

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

953 4.1.1. The <TestItem> Element

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

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

959 <TestOp> element

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

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

```

963 <TestItem objectType="profile">
964   <TestOp//Age >= '21'</TestOp>
965 </TestItem>
966
967

```

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

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

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

972 Service Specific XPath Functions

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

```
976 number profile:age-compare( [//age,] int test-age, string operator )  
977  
978
```

979 and permit selection like

```
980 <TestOp>profile:age-compare( '21', 'gt' )</TestOp>  
981  
982
```

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

986 predefined XML attribute

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

990 4.1.2. The `<QueryItem>` Element

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

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

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

998 The `<QueryItem>` can also have a `<Sort>` element. The type and possible content of this element are specified by
999 the services using this feature. The `<Sort>` element contains the criteria according to which the data in the response
1000 should be sorted. For example, address cards of a contact book could be sorted based on names using either ascending
1001 or descending order. As sorting is resource consuming the service specification MUST use sorting very carefully and
1002 specify sorting only based on the data and criteria which are really needed. In many cases sorting on the server side
1003 is not needed at all. When sorting is needed, only a very limited set of available sorting criteria should be defined.

1004 The `<QueryItem>` can also have a `<ChangeFormat>` element (see [Figure 6](#)). The value of this element specifies, in
1005 which format the requesting WSC would like to have the data, when querying for changes. Two different formats are
1006 defined in this specification. These formats are explained in the processing rules (see [Section 4.4](#)).

1007 The `<QueryItem>` element can have two XML attributes qualifying the query in more detail:

1008 `includeCommonAttributes` (optional) The `includeCommonAttributes` specifies what kind of response is
 1009 requested. The default value is `False`, which means that only the data specified in the
 1010 service definition is returned. If the common XML attributes specified for container and
 1011 leaf elements in this document are also needed, then this XML attribute must be given the
 1012 value `True`. If the `id` XML attribute is used for distinguishing similar elements from one
 1013 other by the service, it **MUST** always be returned, even if the `includeCommonAttributes`
 1014 is `False`.

1015 The `xml:lang` and `script` XML attributes are always returned when they exist.

1016 `changedSince` (optional) The `changedSince` XML attribute should be used when the requestor wants to get only
 1017 the data which has changed since the time specified by this XML attribute. The `changed`
 1018 data can be returned in different ways. A WSC should specify the format it prefers using
 1019 the element **<ChangeFormat>**. Please note that use of this `changedSince` XML attribute
 1020 does not require a service to support the common XML attribute `modificationTime`.
 1021 The service can keep track of the modification times without providing those times as
 1022 `modificationTime` XML attributes for different data elements.

1023 In addition to the `id` XML attribute, the **<ResultQuery>** or **<QueryItem>** element can also have an `itemID` XML
 1024 attribute. The `itemID` XML attribute is correlated with `itemIDRef` XML attributes in the **<Data>** elements in the
 1025 response to match the data to the **<QueryItem>** that produced them. Such correlation is necessary if the **<Query>**
 1026 element contains multiple **<QueryItem>** elements.

1027 4.1.3. Pagination

1028 When the search criteria defined in the **<Select>** matches multiple elements of same type and name, the WSC may
 1029 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
 1030 attributes `count`, `offset`, `setID` and `setReq` of the **<QueryItem>** element. The basic XML attributes are the
 1031 `count` and the `offset`:

1032 `count` (optional) The `count` XML attribute defines, how many elements should returned in a response. This
 1033 is the amount of the elements directly addressed by the **<Select>**, their descendants are
 1034 automatically included in the response, if not elsewhere otherwise specified.

1035 `offset` (optional) The `offset` XML attribute specifies, from which element to continue, when querying for
 1036 more data. The default value is zero, which refers to the first element.

```

1037 <xs:attributeGroup name="PaginationAttributeGroup">
1038   <xs:attribute name="count" use="optional" type="xs:nonNegativeInteger" />
1039   <xs:attribute name="offset" use="optional" type="xs:nonNegativeInteger" default="0" />
1040   <xs:attribute name="setID" use="optional" type="lu:IDType" />
1041   <xs:attribute name="setReq" use="optional">
1042     <xs:simpleType>
1043       <xs:restriction base="xs:string">
1044         <xs:enumeration value="Static" />
1045         <xs:enumeration value="DeleteSet" />
1046       </xs:restriction>
1047     </xs:simpleType>
1048   </xs:attribute>
1049 </xs:attributeGroup>
1050 <xs:attributeGroup name="PaginationResponseAttributeGroup">
1051   <xs:attribute name="remaining" use="optional" type="xs:integer" />
1052   <xs:attribute name="nextOffset" use="optional" type="xs:nonNegativeInteger" default="0" />
1053   <xs:attribute name="setID" use="optional" type="lu:IDType" />
1054 </xs:attributeGroup>
  
```

1055 **Figure 11. XML Attributes for Pagination**

1056 Changes may happen while a WSC is requesting the data in smaller sets as this requires multiple **<Query>** messages
1057 and so will cause multiple **<QueryResponse>**s. This is not a problem for many services, but with some services
1058 this might cause problems as an inconsistent set of data may be returned to the requesting WSC. If supported by
1059 the service type and the WSP, a WSC may request that the modifications done by others are not allowed to effect
1060 what the requesting WSC gets. In the first **<Query>** of a sequence, the requesting WSC includes the `setReq`
1061 XML attribute with the value `Static`. The query response returns an identification for the set and in the following
1062 queries, this is included as the value of the `setID` XML attribute. At the end the WSC requests that the set is deleted
1063 (`setReq="DeleteSet"`) to free the resources on the WSP side.

1064 `setID` (optional) The `setID` XML attribute contains an identification of a set. This must be used by a WSC,
1065 when it wants to make sure that no modifications are done to the set, while it is querying the
1066 data from the set.

1067 `setReq` (optional) With the `setReq` XML attribute a WSC is able to request that a consistent set is created for
1068 coming queries (value `Static`) or a set is deleted (`DeleteSet`).

1069 A service specification MUST specify the elements for which the pagination is supported. The pagination is not meant
1070 to be available for every request, just for a selected types of requests. As the use of the static sets may consume more
1071 resources on the server side than the normal pagination, the use of static sets must be considered carefully.

1072 4.2. The **<QueryResponse>** Element

1073 In addition to different identifiers the **<QueryResponse>** contains

1074 **<Status>** Overall success or failure of the query

1075 **<TestResult>** (optional) Indications of the outcomes of the test items that were present in the **<Query>**.

1076 **<Data>** (optional) The data resulting from **<QueryItem>** elements. Each **<Data>** is correlated to corresponding
1077 **<QueryItem>** using `itemIDRef` XML attribute.

1078 The **<QueryResponse>** elements are correlated, using their `itemIDRef` XML attributes, to the **<Query>** elements
1079 (`ItemID` XML attributes).

1080 The requested data is encapsulated inside **<Data>** elements. One **<Data>** element contains data requested by one
1081 **<QueryItem>** element. If there were multiple **<QueryItem>** elements in the **<Query>**, the **<Data>** elements are
1082 linked to their corresponding **<QueryItem>** elements using the `itemIDRef` XML attributes.

1083 If a WSC requested sorting, but a WSP does not support the requested type of sorting or sorting in general, a WSP
1084 SHOULD return the data unsorted, but then it MUST indicate this by including the XML attribute `notSorted` within
1085 the **<Data>** element carrying the unsorted data. The `notSorted` XML attribute may have either the value `Now`, when
1086 the requested sorting is not supported, but sorting in general is, or `Never`, when the sorting is not supported at all.

1087 If a WSC was querying for changes, the **<Data>** element may contain the XML attribute `changeFormat` to indicate
1088 in which format the changes are returned (see [Figure 6](#)).

1089 The **<Data>** element extends `ItemDataType` with XML attributes `nextOffset` and `remaining`, when a smaller set
1090 of the data instead all the data was requested using the `count` and the `offset` XML attributes in the request. The
1091 `nextOffset` XML attribute in a response is the offset of the first item not included in the response. So the value of
1092 the `nextOffset` XML attribute in a response can be used directly for the `offset` XML attribute in the next request,
1093 when the data is fetched sequentially. The `remaining` XML attribute defines, how many items there are after the last
1094 item included in the response. The `setID` XML attribute is also included, when a static set is accessed.

1095 If there were multiple **<Query>** elements in the request message, the **<QueryResponse>** elements are linked to
1096 corresponding **<Query>** elements with `itemIDRef` XML attributes.

1097 4.3. <ResultQuery> or <QueryItem> Conditioned by <TestItem>

1098 ResultQueryType has itemIDRef and contingency attributes so that the query items can be made contingent on
1099 some <TestItem>. This itemIDRef correlates with the itemID in the <TestItem>, see [Section 4.1.1](#)

1100 1. A service specification MAY restrict, or forbid, use of <TestItem> in conjunction with <ResultQuery> or
1101 <QueryItem>. If use of <TestItem> is fully supported, the WSP MAY register the discovery option keyword

1102 urn:liberty:dst:contingentQueryItems

1103
1104

1105 2. If contingency attribute is present, then itemIDRef MUST be present as well and vice versa.

1106 3. If the itemIDRef attribute does not match <TestItem> then the WSP MUST stop processing the <QueryItem>
1107 or <ResultQuery> and return a second level status code NoSuchTest.

1108 4. If <QueryItem> or <ResultQuery> has a contingency attribute, the WSP MUST process the <QueryItem>
1109 or <ResultQuery> if and only if the <TestItem> referenced using the itemIDRef evaluates to the value of the
1110 contingency XML attribute.

1111 5. The scope of the itemIDRef is one <Query>, <Create>, or <Modify>. itemIDRef MUST NOT refer to
1112 itemID in another top level element. The itemID XML attributes of <TestItem> elements MUST be unique
1113 within one <Query>, <Create>, or <Modify> element in the request. The <TestItem>, <ResultQuery>, and
1114 <QueryItem> share same itemID space.

1115 4.4. Processing Rules for Queries

1116 NOTE: The common processing rules specified earlier MUST also be followed (see [Section 3](#)).

1117 4.4.1. Processing Rules for Multiple <QueryItem> Elements

1118 One <Query> element can contain multiple <QueryItem> elements. The following rules specify how those must be
1119 supported and handled:

1120 1. A WSP MUST support one <QueryItem> element inside a <Query> and SHOULD support multiple. If a WSP
1121 supports only one <QueryItem> element inside a <Query> and the <Query> contains multiple <QueryItem>
1122 elements, the processing of the whole <Query> MUST fail and a status code indicating failure MUST be returned
1123 in the response. A more detailed status code with the value NoMultipleAllowed SHOULD be used in addition
1124 to the top level status code. If a WSP supports multiple <QueryItem> elements inside a <Query>, it MAY
1125 register the urn:liberty:dst:multipleQueryItems discovery option keyword.

1126 2. If the <Query> contains multiple <QueryItem> elements, the WSP MUST add itemID XML attributes to each
1127 <QueryItem> element. The WSP MUST link the <Data> elements to corresponding <QueryItem> elements
1128 using the itemIDRef XML attributes, if there were itemID XML attributes in the <QueryItem> elements
1129 and there were multiple <QueryItem> elements in the <Query>. The itemIDRef XML attribute in a <Data>
1130 element MUST have the same value as the itemID XML attribute in the corresponding <QueryItem> element.

1131 3. If processing of a <QueryItem> fails, any remaining unprocessed <QueryItem> elements SHOULD NOT be
1132 processed. The data for the already processed <QueryItem> elements SHOULD be returned in the response
1133 message and the status code MUST indicate the failure to completely process the whole <Query>. A more
1134 detailed status SHOULD be used in addition to the top level status code to indicate the reason for failing to
1135 process the first failed <QueryItem>.

1136 4. Unless service specification expressly allows an empty <Query/>, <Query> MUST have at least one
1137 <QueryItem> or <TestItem> element. If not, <Query> MUST fail with EmptyRequest second level code. If
1138 empty <Query/> is allowed, it SHOULD have semantics of returning the default document.

1139 4.4.2. Processing Rules for <Select> Element

- 1140 1. If there is no `changedSince` XML attribute in the <QueryItem> element and the <Select> requests valid data
1141 elements, but there are no values, the WSP MUST NOT return any <Data> element for that <QueryItem>
1142 unless a WSC is requesting pagination. In this case a WSP MUST return the <Data> element containing the
1143 remaining and the `nextOffset` XML attributes even, when no actual data is returned (see processing rules
1144 related to pagination later on).
- 1145 2. If the <Select> requests multiple data elements, the WSP MUST return all of those data elements inside the
1146 <Data> element corresponding to the containing <QueryItem>.

1147 4.4.3. Sorting Query Results

- 1148 1. When the <Sort> element is included in a <QueryItem> element, the data returned inside a <Data> element
1149 SHOULD be sorted according to the criteria given in the <Sort> element. If a WSP doesn't support sorting, it
1150 SHOULD return the requested data unsorted. When the data is returned unsorted, the `notSorted` XML attribute
1151 MUST be used in the <Data> element containing the unsorted data. A WSP MAY also choose to fail to process
1152 the <QueryItem>, if it does not support sorting. In that case the second level status code `SortNotSupported`
1153 SHOULD be used in addition to the top level status code. A WSP may also register discovery option keyword
1154 urn:liberty:dst:noSorting, if the sorting has been specified for the service type, but the WSP doesn't support it.
- 1155 2. If the content of the <Sort> element is not according to service specifications, a WSP SHOULD return the
1156 requested data unsorted. When the data is returned unsorted, the `notSorted` XML attribute MUST be used in
1157 the <Data> element containing the unsorted data and the second level status code `InvalidSort` SHOULD also
1158 be used. A WSP MAY also choose to fail to process the <QueryItem>, if the content of the <Sort> element
1159 is not according to service specifications. In this kind of a case the second level status code `InvalidSort`
1160 SHOULD be used in addition to the top level status code. If the content of the <Sort> element is valid, but a
1161 WSP does not support the requested type of sorting, it SHOULD return the requested data unsorted. When the
1162 data is returned unsorted, the `notSorted` XML attribute MUST be used in the <Data> element containing the
1163 unsorted data. A WSP MAY also choose to fail to process of the <QueryItem>, if it does not support the
1164 requested type of sorting. It SHOULD use the second level status code `RequestedSortingNotSupported` in
1165 addition to the top level status code.
- 1166 3. When the `notSorted` XML attribute is used, it MUST have the value `Now`, when a WSP supports sorting, but
1167 not the requested type or the content of the <Sort> element was invalid. The `notSorted` XML attribute MUST
1168 have the value `Never`, when a WSP does not support sorting at all.

1169 4.4.4. Pagination of Query Results

1170 A WSC may want to receive the data in smaller sets instead of getting all the data at once, when there can be many
1171 elements with the same name. A WSC indicates this using either or both of the XML attributes `count` and `offset`
1172 in a <QueryItem> element, when the <Select> addresses a set of elements all having the same name. The number of
1173 elements inside this set may be restricted further by other parameters. Also access rights and policies may reduce the
1174 set of elements a WSC is allowed to get.

- 1175 1. A WSP MUST always follow the same ordering, when the <Select> and <Sort> elements have the same values
1176 and either or both of XML attributes `count` and `offset` are used in the <QueryItem> element. If same query
1177 is made twice without a modification intervening, the result set MUST be the same and in same order. This is
1178 needed to make sure, for example, that a WSC really gets the next ten items, when asking for them, and not e.g.
1179 five of the previously returned items with five new items.

- 1180 2. When either or both of the XML attributes `count` and `offset` is used in a **<QueryItem>** element and a WSP
1181 doesn't support pagination, the processing of whole **<QueryItem>** element **MUST** fail and the second level
1182 status code `PaginationNotSupported` **SHOULD** be used in addition to the top level status code. A WSP may
1183 support pagination, but not for the requested elements. In such a case the processing of whole **<QueryItem>**
1184 element **MUST** fail and the second level status code `RequestedPaginationNotSupported` **SHOULD** be used
1185 in addition to the top level status code. If a WSP doesn't support pagination at all, it **MAY** register the discovery
1186 option keyword `urn:liberty:dst:noPagination` to indicate this.
- 1187 3. When the `count` XML attribute is included in a **<QueryItem>** element, the corresponding **<Data>** element in
1188 the **<QueryResponse>** **MUST NOT** contain more elements addressed with the value of the **<Select>** element
1189 than specified by the `count` XML attribute. A WSP **MAY** return a smaller number of elements of the same
1190 name that requested by a WSC. If the `count` XML attribute has the value zero, the WSP **MUST NOT** return any
1191 data elements inside the **<Data>** element. This `count="0"` may be used for querying the number of remaining
1192 elements starting from the specified offset, e.g., from offset zero, i.e., the total number of the elements addressed
1193 by the **<Select>** element. When the `count` XML attribute is not used in a **<QueryItem>** element, it means that
1194 the WSC requests for all data specified by other parameters like the **<Select>** element starting from the specified
1195 offset. As the default value for the `offset` XML attribute is zero, the case when neither of the XML attributes
1196 `offset` or `count` is not present reduces to a normal query.
- 1197 4. When pagination is requested by a WSC, the elements inside a **<Data>** element **MUST** be in the ascending
1198 order of their offsets. The first element **MUST** have the offset specified by the `offset` XML attribute in the
1199 **<QueryItem>** element. The **<Data>** element **MUST** have both XML attributes `nextOffset` and `remaining`.
1200 The `nextOffset` XML attribute **MUST** have the offset of the first element not returned in the response. The
1201 value of the `remaining` XML attribute **MUST** define how many elements there are left starting from the value of
1202 the `nextOffset`, if a WSP knows that (e.g., that information might not be available from a backend system). If
1203 WSP does not know the exact value, it **MUST** use the value `-1` for the `remaining` XML attribute until it knows
1204 the value or there is no data left (`remaining="0"`). When `remaining="-1"`, a WSC must make new requests
1205 until `remaining="0"`, if it wants to get all the data.
- 1206 5. Usually, when there is no data matching the different query parameters, no **<Data>** element is returned in a
1207 **<QueryResponse>**. When either or both of the `count` and `offset` attributes are used, the **<Data>** element
1208 **MUST** be returned, even, when no data is returned (e.g., no data available or `count="0"` used to get the number
1209 of data items). This is required so that a WSP can return the `remaining` and the `nextOffset` XML attributes
1210 to the requesting WSC.
- 1211 6. When the `setReq` XML attribute is included in a **<QueryItem>** element and has the value `Static`, the WSP
1212 **SHOULD** return the `setID` XML attribute to the requesting WSC and process **<QueryItem>** elements later
1213 having this `setID` based on the data the WSP has at the time, when the value for the `setID` was created. If
1214 a WSP receives a **<QueryItem>** element having the `setReq` XML attribute and does not support static sets for
1215 the requested data or not at all, the processing of the **<QueryItem>** element **MUST** fail and a second level status
1216 code `StaticNotSupported` **SHOULD** be used in addition to the top level status code. If a WSP doesn't support
1217 static sets at all, it **MAY** register the discovery option keyword `urn:liberty:dst:noStatic`.
- 1218 7. When the `setID` XML attribute is included in a request, the following parameters **MUST NOT** be used in
1219 a **<QueryItem>** element: the **<Select>** element, the **<Sort>** element, the `changedSince` XML attribute, the
1220 `includeCommonXML Attributes` XML attribute, or the `predefined` XML attribute. The requests are made
1221 from an earlier defined static set and the `count` and the `offset` XML attributes are used to define, what is
1222 requested from that set. If any of the mentioned parameters is present, when the `setID` XML attribute is used,
1223 it is unclear what a WSC wants and the processing of the whole **<QueryItem>** **MUST** fail and a second level
1224 status code `SetOrNewQuery` **SHOULD** be used in addition to the top level status code.
- 1225 8. When the `setID` XML attribute is included in a **<QueryItem>** element and has a valid value, the **<Data>** element
1226 in the response **MUST** always have the `setID` XML attribute.

- 1227 9. When a static set is created, the requesting WSC SHOULD query all the data it needs from this set as soon as
1228 possible and delete the static set immediately after this using `setReq="DeleteSet"`. A WSP MAY also delete
1229 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
1230 a non-existing static set, the processing of the whole `<QueryItem>` MUST fail and the second level status code
1231 `InvalidSetID` SHOULD be used in addition to the top level status code.
- 1232 10. The `setReq="Static"` and the `setID` XML attribute MUST NOT be used simultaneously in a `<QueryItem>`
1233 element. If they are used, the WSP MUST ignore the `setReq="Static"` and process the `<QueryItem>` element
1234 like the `setReq` XML attribute would not be present.
- 1235 11. If the `setReq` XML attribute has some other value than `Static` or `DeleteSet`, the processing of the whole
1236 `<QueryItem>` element must fail and a second level status code `InvalidSetReq` SHOULD be used in addition
1237 to the top level status code.

1238 4.4.5. Effect of Access and Privacy Policies

1239 Even when the requested data exists, it should be noted that access and privacy policies specified by the resource owner
1240 may cause the request to result in data not being returned to the requestor.

1241 When a WSP processes a `<QueryItem>`, it MUST check whether the resource owner (the Principal, for example) has
1242 given consent to return the requested information. To be able to check WSC specific access rights, the WSP MUST
1243 authenticate the WSC (see [[LibertySecMech](#)]). The WSP MUST also check that any usage directive given in the
1244 request is acceptable based on the usage directives defined by the resource owner (see [[LibertySOAPBinding](#)]). If
1245 either check fails for any piece of the requested data, the WSP MUST NOT return that piece of data. Note that there
1246 can be consent for returning some data element, but not its XML attributes. For example, a resource owner might not
1247 want to release the `modifier` XML attribute, if she does not want to reveal information about which services she uses.
1248 The data for which there is no consent from the resource owner MUST be handled as if there was no data. The WSP
1249 MAY try to get consent from the resource owner while processing the request, e.g., by using an interaction service,
1250 see [[LibertyInteract](#)]. A WSP might check the access rights and policies in usage directives at a higher level, before
1251 getting to DST processing and MAY, in this case, just return an ID-* Fault Message [[LibertySOAPBinding](#)] without
1252 processing the `<Query>` element at all, if the requesting WSC is not allowed to access the data.

1253 4.4.6. Querying Changes Since Specified Time

1254 It is possible to query changes since a specified time using the `changedSince` XML attribute.

- 1255 1. If the `<QueryItem>` element contains the `changedSince` XML attribute, the WSP SHOULD return only those
1256 elements addressed by the `<Select>` which have been modified since the time specified in the `changedSince`
1257 XML attribute. There are two different formats, in which the changed data can be returned. A WSC SHOULD
1258 indicate using the `<ChangeFormat>` element the format it prefers and also, if it understands the other format.
1259 The two formats are `ChangedElements` and `CurrentElements`. If a service specification doesn't specify
1260 anything else the value `ChangedElements` MUST be used as a default value as it is compatible with the format
1261 used in the version 1.0 of the Data Services Template.
- 1262 2. A WSP MUST ignore the `<ChangeFormat>` element, if the `changedSince` XML attribute is not used in the
1263 same `<QueryItem>` element. A WSP MUST NOT use a format, which a WSC does not understand. Note that
1264 format `ChangedElements`, has the format `All` as a fallback solution, when a WSP doesn't have all the needed
1265 change history information. Also if a WSP doesn't support requesting only changed data, it returns all data.
- 1266 3. A `<QueryItem>` element MAY contain two `<ChangeFormat>` element with different values. A WSP SHOULD
1267 use the format specified by the first `<ChangeFormat>` element, but, if it does not support that format, it MAY
1268 use the format specified by the second `<ChangeFormat>` element.

- 1269 4. If a WSP does not support the format a WSC is requesting to be used, the processing of the **<QueryItem>** MUST
1270 fail and the second level status code `FormatNotSupported` SHOULD be used in addition to the top level status
1271 code.
- 1272 5. If a WSC requests the `ChangedElements` format and a WSP supports it, the WSP SHOULD return only the
1273 changed information. If some element has been deleted, a WSP SHOULD return an empty element to indicate
1274 the deletion (**<ElementName+/>**). The only allowed exception to this is that the WSP does not have enough
1275 history information available to be able to return only the changed information. In that case it MUST use format
1276 `All` and return all current elements with their values even if those have not changed since the specified time.
- 1277 6. If a WSC requests the `CurrentElements` format and a WSP supports it, the WSP SHOULD return only the
1278 currently existing elements. It SHOULD return an empty element, if the element has not changed, to indicate that
1279 no change has happened (**<ElementName/>**).
- 1280 N.B. As empty elements are used to indicate either deleted or not changed elements depending on the used format,
1281 the formats `CurrentElements` and `ChangedElements` do not work well, if the data hosted by a service may
1282 contain empty elements. In those cases a service should either use only format `All` or always have some XML
1283 attributes for the otherwise empty elements.
- 1284 7. If a WSC has used the **<ChangeFormat>** element in a request, a WSP MUST use the `changeFormat` XML
1285 attribute in the response to indicate, which format is used. A WSP MUST not use the `changeFormat` XML
1286 attribute in a response, if the **<ChangeFormat>** element was not used in the corresponding request so the
1287 processing stays version 1.0 compatible, when the **<ChangeFormat>** element is not used.
- 1288 8. If there can be multiple elements with same name, the `id` XML attribute or some other XML attribute used to
1289 distinguish the elements from each other MUST be included (e.g., in case of an ID-SIS Personal Profile service
1290 the following empty element could be returned **<AddressCard id="tr7632q"/>** to indicate a deleted or not
1291 changed **<AddressCard>** depending on the used format). If the value of the `id` XML attribute or some other
1292 XML attribute used for distinguishing elements with same name is changed, the WSP MUST consider this as a
1293 case, in which the element with the original value of the distinguishing XML attribute is deleted and a new one
1294 with the new value of the distinguishing XML attribute is created. To avoid this, a WSP MAY refuse to accept
1295 modifications of a distinguishing XML attribute and MAY require that an explicit deletion of the element is done
1296 and a new one created.
- 1297 9. If the elements addressed by the **<Select>** have some values, but there has been no changes since the time specified
1298 in the `changedSince` XML attribute, the WSP MUST return empty **<Data>** element (**<Data/>**), when it returns
1299 the changes properly. This empty **<Data>** element indicates that no changes have occurred. There might be cases
1300 in which the WSP is not able to return changes properly, see later processing rules. Please note that in cases that
1301 have no values, no **<Data>** element is returned to indicate this. So empty **<Data>** element has different semantics
1302 than missing **<Data>** element.
- 1303 10. If the **<QueryItem>** element contains the `changedSince` XML attribute and a WSP is not keeping track of
1304 modification times, it SHOULD process the **<QueryItem>** element as there would be no `changedSince` XML
1305 attribute, and indicate this in the response using the second level status code `ChangedSinceReturnsAll`. This
1306 is not considered a failure and the rest of the **<QueryItem>** elements MUST be processed. Also it might be
1307 that a WSP does not have a full change history and so for some queries, it is not possible to find out, which
1308 changes occurred after the specified time. As processing with access rights and policy in place might be quite
1309 complex, a WSP might sometimes process the query for changes properly and sometime process it as if there
1310 were no `changedSince` XML attribute. In those cases, when a WSP returns all current values, it SHOULD
1311 indicate this with the second level status code `AllReturned` and, if the **<ChangeFormat>** element was used
1312 in the request, the `changeFormat` XML attribute with the value `All` SHOULD be used. This is also not
1313 considered a failure and the rest of the **<QueryItem>** elements MUST be processed. Please note that the status
1314 code `AllReturned` differs from the status code `ChangedSinceReturnsAll`, as `ChangedSinceReturnsAll`
1315 means that the WSP never processes the `changedSince` XML attribute properly. A WSP MUST use either
1316 `AllReturned` or `ChangedSinceReturnsAll` as the second level status code, when it returns data, but does

1317 not process the `changedSince` XML attribute properly, i.e., returns only the changes. If a WSP will not process
1318 the `<QueryItem>` elements with a `changedSince` XML attribute at all, it MUST indicate this with top level
1319 status code `Failed` and SHOULD also return a second level status code of `ChangeHistoryNotSupported`
1320 in the response. In this case a WSP MUST NOT return any `<Data>` element for the `<QueryItem>` element
1321 containing the `changedSince` XML attribute. If a WSP processes the `changedSince` XML attribute, it
1322 MUST also support the `notChangedSince` XML attribute for `<ModifyItem>` element and MAY register the
1323 `urn:liberty:dst:changeHistorySupported` discovery option keyword. Please note that still in some cases a WSP
1324 MAY return `AllReturned`.

1325 11. Access rights and policies in place may affect how the queries for changes can work as they affect which elements
1326 and XML attributes a WSC is allowed to see. If a WSC was originally allowed to get the requested data, but is
1327 no longer after some change in access policies, then from its point of view that data is deleted and that should
1328 be taken into account in the response. If the WSP notices that access rights have changed, and the current rights
1329 do not allow access, it MUST return all data except the data for which the access rights were revoked, and use
1330 the second level status code `AllReturned` and, if the `<ChangeFormat>` element was used in the request, the
1331 `changeFormat` XML attribute with the value `All` SHOULD be used. The WSP MUST NOT return empty
1332 elements for the data for which access rights were changed even if the format `ChangedElement` was requested,
1333 as this might reveal the fact that this specific data has at least existed at the service in some point of time. Please
1334 note that it might be the case that the data was added after the WSCs access rights were revoked and the WSC was
1335 never supposed to be aware of the existence of that data. If the WSP notices that the access rights are changed
1336 and the current rights do allow access, it MUST consider the data for which the access rights are changed, as if it
1337 were just created.

1338 12. Both the WSC and WSP may have policies specified by the Principal for control of their data. Only by comparing
1339 policy statements made by the WSC (via `<UsageDirective>` elements (see [\[LibertySOAPBinding\]](#)) with policies
1340 maintained on behalf of the Principal by the WSP it is possible to fully determine the effects of interaction
1341 between these sets of policies. As it might be too expensive to search for policies the WSC promised to honor
1342 when it made the original request, and this information might not even be available, the WSP might be only
1343 capable of making the decision based on the policy changes made by the Principal. If some data is prevented
1344 from being returned to the WSC due to conflicts in policies and the WSP notices that the Principal's policies have
1345 changed, it MUST return all data except that for which the Principal's policy has denied access against the current
1346 policy of a requesting WSC, and use the second level status code `AllReturned` to indicate that the WSC must
1347 check the response carefully to find out what has changed. Also if the `<ChangeFormat>` element was used in the
1348 request, the `changeFormat` XML attribute with the value `All` SHOULD be used. The WSP MUST NOT return
1349 empty elements for the data for which the Principal's policy was changed even if the format `ChangedElements`
1350 was requested, as this might reveal the fact that this specific data was exposed by the service at some point in
1351 time. Please note that it might be the case that that data has been added after the policies were changed and the
1352 requesting WSC was never supposed to be aware of that data, unless it changed the policy it promises to honor.
1353 If the WSP notices that the Principal's policy has changed and the current policy does allow access, it MUST
1354 consider the data for which the policy is changed as if it had been just created. If a WSC changes the policy it
1355 promises to honor, it SHOULD make a new query without a `changedSince` XML attribute.

1356 13. As mentioned earlier, the WSP might in some cases return all the current data the `<Select>` points to, and not just
1357 the changes using specified format, even when the `changedSince` XML attribute is present. So the WSC MUST
1358 compare the returned data to previous data it had queried earlier to find out what really has changed. Note that
1359 this MUST be done even when the WSP has processed the `changedSince` correctly, because some values might
1360 have been changed back and forth and now they have same values that they used to have earlier, despite the most
1361 current previous values being different.

1362 4.4.7. Requesting Common XML Attributes

1363 The common XML attributes are not always returned. A WSC may indicate with the `includeCommonAttributes`
1364 XML attribute, whether it wants to have the common XML attributes or not.

- 1365 1. If the `includeCommonAttributes` is set to `True`, the common XML attributes specified by XML at-
 1366 tribute groups `commonAttributes` and `leafAttributes` MUST be included in the response, if their val-
 1367 ues are specified for the requested data elements. The ACC XML attributes MAY be left out, if the value is
 1368 urn:liberty:dst:acc:unknown.
- 1369 2. If the `id` XML attribute is used for distinguishing similar elements from each other by the service, it MUST be
 1370 returned, even if the `includeCommonAttributes` is false. Also, when either or both of the XML attributes
 1371 `xml:lang` and `script` are present, they MUST be returned, even if the `includeCommonAttributes` is false

1372 4.5. Examples

1373 The following query example, based on hypothetical profile service, requests the common name and home address of
 1374 a Principal:

```
1375 <hp:Query xmlns:hp="urn:liberty:hp:2005-07">
1376 <hp:QueryItem itemID="name">
1377 <hp:Select>/hp:HP/hp:CommonName</hp:Select>
1378 </hp:QueryItem>
1379 <hp:QueryItem itemID="home">
1380 <hp:Select>
1381 /hp:HP/hp:AddressCard
1382 [hp:AddressType="urn:liberty:id-sis-hp:addrType:home" ]
1383 </hp:Select>
1384 </hp:QueryItem>
1385 </hp:Query>
1386
1387
```

1388 This query may generate the following response:

```
1389 <hp:QueryResponse xmlns:hp="urn:liberty:hp:2005-07">
1390 <hp:Status code="OK"/>
1391 <hp:Data itemIDRef="name">
1392 <hp:CommonName>
1393 <hp:CN>Zita Lopes</hp:CN>
1394 <hp:AnalyzedName nameScheme="firstlast">
1395 <hp:FN>Zita</hp:FN>
1396 <hp:SN>Lopes</hp:SN>
1397 <hp:PersonalTitle>Dr.</hp:PersonalTitle>
1398 </hp:AnalyzedName>
1399 <hp:AltCN>Maria Lopes</hp:AltCN>
1400 <hp:AltCN>Zita Maria Lopes</hp:AltCN>
1401 </hp:CommonName>
1402 </hp:Data>
1403 <hp:Data itemIDRef="home">
1404 <hp:AddressCard id='9812'>
1405 <hp:AddressType>
1406 urn:liberty:id-sis-hp:addrType:home
1407 </hp:AddressType>
1408 <hp:Address>
1409 <hp:PostalAddress>
1410 c/o Carolyn Lewis$2378 Madrona Beach Way North
1411 </hp:PostalAddress>
1412 <hp:PostalCode>98503-2341</hp:PostalCode>
1413 <hp:L>Olympia</hp:L>
1414 <hp:ST>wa</hp:ST>
1415 <hp:C>us</hp:C>
1416 </hp:Address>
1417 </hp:AddressCard>
1418 </hp:Data>
1419 </hp:QueryResponse>
1420
1421
1422
```

1423 If there was no user consent for the release of the **<hp:CommonName>** or for the whole **<hp:AddressCard>** with
1424 [hp:AddressType="urn:liberty:id-sis-hp:addrType:home"], apart from the country information, then
1425 the response is as follows (including a timestamp, as this service supports change history):

```
1426 <hp:QueryResponse
1427     xmlns:hp="urn:liberty:hp:2005-07"
1428     timeStamp="2003-02-28T12:10:12Z">
1429   <hp:Status code="OK"/>
1430   <hp:Data itemIDRef="home">
1431     <hp:AddressCard id='9812'>
1432       <hp:AddressType>
1433         urn:liberty:id-sis-hp:addrType:home
1434       </hp:AddressType>
1435       <hp:Address><hp:C>us</hp:C></hp:Address>
1436     </hp:AddressCard>
1437   </hp:Data>
1438 </hp:QueryResponse>
1439
1440
```

1441 If there was no **<hp:CommonName>** and no **<hp:AddressCard>** with [hp:AddressType="urn:liberty:id-
1442 sis-hp:addrType:home"], then the response is:

```
1443 <hp:QueryResponse
1444     xmlns:hp="urn:liberty:hp:2005-07"
1445     timeStamp="2003-02-28T12:10:12Z">
1446   <hp:Status code="OK"/>
1447 </hp:QueryResponse>
1448
1449
```

1450 The following request queries the fiscal identification number of the Principal with the common XML attributes:

```
1451 <hp:Query xmlns:hp="urn:liberty:hp:2005-07">
1452   <hp:QueryItem includeCommonAttributes="True">
1453     <hp:Select>/hp:HP/hp:LegalIdentity/hp:VAT</hp:Select>
1454   </hp:QueryItem>
1455 </hp:Query>
1456
1457
```

1458 This query may generate the following response:

```
1459 <hp:QueryResponse
1460     xmlns:hp="urn:liberty:hp:2005-07"
1461     id="12345"
1462     timeStamp="2003-05-28T23:10:12Z">
1463   <hp:Status code="OK"/>
1464   <hp:Data>
1465     <hp:VAT
1466       modifier="http://www.accountingservices.com"
1467       modificationTime="2003-04-25T15:42:11Z"
1468       ACC="urn:liberty:dst:acc:secondarydocuments">
1469       <hp:IDValue
1470         modifier="http://www.accountingservices.com"
1471         modificationTime="2003-04-25T15:42:11Z"
1472         ACC="urn:liberty:dst:acc:secondarydocuments">
1473         502677123
1474       </hp:IDValue>
1475       <hp:IDType
1476         modifier="http://www.accountingservices.com"
1477         modificationTime="2003-03-12T09:12:09Z"
1478         ACC="urn:liberty:dst:acc:secondarydocuments">
1479         urn:liberty:altIDType:itcif
1480       </hp:IDType>
1481     </hp:VAT>

```

```

1482     </hp:Data>
1483 </hp:QueryResponse>
1484 <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmlsig#">
1485     . . .
1486 </ds:Signature>
1487
1488
  
```

1489 The following request queries for address information which has been changed since 12:10:12 28 February 2003 UTC:

```

1490 <hp:Query xmlns:hp="urn:liberty:hp:2005-07">
1491   <hp:QueryItem changedSince="2003-02-28T12:10:12Z">
1492     <hp:Select>/hp:HP/hp:AddressCard</hp:Select>
1493   </hp:QueryItem>
1494 </hp:Query>
1495
1496
  
```

1497 This query can generate following response:

```

1498 <hp:QueryResponse
1499   xmlns:hp="urn:liberty:hp:2005-07"
1500   timeStamp="2003-05-30T16:10:12Z">
1501   <hp:Status code="OK"/>
1502   <hp:Data>
1503     <hp:AddressCard id='9812'>
1504       <hp:Address>
1505         <hp:PostalAddress>
1506           2891 Madrona Beach Way North
1507         </hp:PostalAddress>
1508       </hp:Address>
1509     </hp:AddressCard>
1510     <hp:AddressCard id='w1q2' />
1511   </hp:Data>
1512 </hp:QueryResponse>
1513
1514
  
```

1515 Please note that only the changed information inside the **<hp:AddressCard>** is returned. The response shows that
 1516 after the specified time, there was also another **<hp:AddressCard>** present, but that has been deleted. As there can
 1517 be many **<hp:AddressCard>** elements, the id XML attribute is returned to distinguish distinct elements.

1518 If there have been no changes since the specified time, then the response is just:

```

1519 <hp:QueryResponse
1520   xmlns:hp="urn:liberty:hp:2005-07"
1521   timeStamp="2003-05-30T16:10:12Z">
1522   <hp:Status code="OK"/>
1523   <hp:Data/>
1524 </hp:QueryResponse>
1525
1526
  
```

1527 If the same request for changed addresses is made including the **<hp:ChangeFormat>** element:

```

1528 <hp:Query xmlns:hp="urn:liberty:hp:2005-07">
1529   <hp:QueryItem changedSince="2003-02-28T12:10:12Z">
1530     <hp:Select>/hp:HP/hp:AddressCard</hp:Select>
1531     <hp:ChangeFormat>CurrentElements</hp:ChangeFormat>
1532   </hp:QueryItem>
1533 </hp:Query>
1534
1535
  
```

1536 All the current elements are returned in the response:

```

1537 <hp:QueryResponse
1538     xmlns:hp="urn:liberty:hp:2005-07"
1539     timeStamp="2003-05-30T16:10:12Z">
1540 <hp:Status code="OK"/>
1541 <hp:Data changeFormat="CurrentElements">
1542   <hp:AddressCard id='9812'>
1543     <hp:Address>
1544       <hp:PostalAddress>2891 Madrona Beach Way North</hp:PostalAddress>
1545       <hp:PostalCode/>
1546       <hp:L/>
1547       <hp:ST/>
1548       <hp:C/>
1549     </hp:Address>
1550   </hp:AddressCard>
1551 </hp:Data>
1552 </hp:QueryResponse>
1553
1554
  
```

1555 Please note that now all the current elements inside the **<hp:AddressCard>** are returned. The deleted
 1556 **<hp:AddressCard>** is not shown at all and for the elements, which have not changed - only empty elements
 1557 are returned.

1558 If a WSP does not support change history, then the response could be:

```

1559 <hp:QueryResponse
1560     xmlns:hp="urn:liberty:hp:2005-07"
1561     timeStamp="2003-05-30T16:10:12Z">
1562 <hp:Status code="OK">
1563   <Status code="ChangeSinceReturnsAll"/>
1564 </hp:Status>
1565 <hp:Data changeFormat="All">
1566   <hp:AddressCard id='9812'>
1567     <hp:AddressType>urn:liberty:id-sis-hp:addrType:home</hp:AddressType>
1568     <hp:Address>
1569       <hp:PostalAddress>2891 Madrona Beach Way North</hp:PostalAddress>
1570       <hp:PostalCode>98503-2341</hp:PostalCode>
1571       <hp:L>Olympia</hp:L>
1572       <hp:ST>wa</hp:ST>
1573       <hp:C>us</hp:C>
1574     </hp:Address>
1575   </hp:AddressCard>
1576 </hp:Data>
1577 </hp:QueryResponse>
1578
1579
  
```

1580 The rest of the examples are related to pagination and sorting based on fictional address service, so all the DST
 1581 elements in the namespace of that fictional address service.

1582 Parameters **<Select>** and **<Sort>** and returned **<Data>** elements do not have valid contents in the examples as the
 1583 main point is to show the principle how pagination works and the use of the pagination related XML attributes

1584 A Resource contains 40 address cards and the WSC A wants to list those ordered by the City and 10 at the time. Due
 1585 to access rights and policies the WSC A is allowed to get only 30 of those AddressCards. The WSC A makes the first
 1586 query:

```

1587 <ads:Query xmlns:ads="http://www.example.com/2010/12/Addr">
1588   <ads:QueryItem count="10">
1589     <ads:Select>Pointing to the AddressCards</ads:Select>
1590     <ads:Sort>Requesting sorting by the City</ads:Sort>
1591   </ads:QueryItem>
1592 </ads:Query>
  
```

1593
1594

1595 and gets back the first ten address cards ordered by the City:

```
1596 <ads:QueryResponse
1597     xmlns:ads="http://www.example.com/2010/12/Addr"
1598     timeStamp="2004-03-23T03:40:00Z">
1599     <ads:Status code="OK"/>
1600     <ads:Data remaining="20" nextOffset="10">first ten address cards</ads:Data>
1601 </ads:QueryResponse>
1602
1603
```

1604 Then it queries the next ten starting from offset 10:

```
1605 <ads:Query xmlns:ads="http://www.example.com/2010/12/Addr">
1606     <ads:QueryItem count="10" offset="10">
1607         <ads:Select>Pointing to the AddressCards</ads:Select>
1608         <ads:Sort>Requesting sorting by the City</ads:Sort>
1609     </ads:QueryItem>
1610 </ads:Query>
1611
1612
```

1613 and gets those

```
1614 <ads:QueryResponse
1615     xmlns:ads="http://www.example.com/2010/12/Addr"
1616     timeStamp="2004-03-23T03:40:20Z">
1617     <ads:Status code="OK"/>
1618     <ads:Data remaining="10" nextOffset="20">next ten address cards</ads:Data>
1619 </ads:QueryResponse>
1620
1621
```

1622 After this the WSC B adds one more address card to the resource. The WSC A is allowed to get this address card, but
1623 when sorting based on the City, this new card has the offset 15. When the WSC A fetches the next ten address cards:

```
1624 <ads:Query xmlns:ads="http://www.example.com/2010/12/Addr">
1625     <ads:QueryItem count="10" offset="20">
1626         <ads:Select>Pointing to the AddressCards</ads:Select>
1627         <ads:Sort>Requesting sorting by the City</ads:Sort>
1628     </ads:QueryItem>
1629 </ads:Query>
1630
1631
```

1632 It gets ten address cards, but it has already received the first address card already in the previous response.

```
1633 <ads:QueryResponse
1634     xmlns:ads="http://www.example.com/2010/12/Addr"
1635     timeStamp="2004-03-23T03:41:00Z">
1636     <ads:Status code="OK"/>
1637     <ads:Data remaining="1" nextOffset="30">next ten address cards</ads:Data>
1638 </ads:QueryResponse>
1639
1640
```

1641 Finally the WSC A fetches the last address card.

```
1642 <ads:Query xmlns:ads="http://www.example.com/2010/12/Addr">
1643     <ads:QueryItem count="1" offset="30">
1644         <ads:Select>Pointing to the AddressCards</ads:Select>
1645         <ads:Sort>Requesting sorting by the City</ads:Sort>
```

```
1646     </ads:QueryItem>
1647 </ads:Query>
1648
1649
```

1650 and gets the 31st address card from offset 30.

```
1651 <ads:QueryResponse
1652     xmlns:ads="http://www.example.com/2010/12/Addr"
1653     timeStamp="2004-03-23T03:41:17Z">
1654   <ads:Status code="OK"/>
1655   <ads:Data remaining="0" nextOffset="31">the last address card</ads:Data>
1656 </ads:QueryResponse>
1657
1658
```

1659 So the WSC A didn't get this new address card added by the WSC B and got one card twice.

1660 In an alternative scenario, if supported by the WSP, the WSC A requests a static set to be created so that simultaneous
1661 modifications can not affect the results the WSC A gets. The initial request includes the `setReq` XML attribute:

```
1662 <ads:Query xmlns:ads="http://www.example.com/2010/12/Addr">
1663   <ads:QueryItem count="10" setReq="Static">
1664     <ads:Select>Pointing to the AddressCards</ads:Select>
1665     <ads:Sort>Requesting sorting by the City</ads:Sort>
1666   </ads:QueryItem>
1667 </ads:Query>
1668
1669
```

1670 In the response the first ten address cards are returned together with a handle to this static set (the XML attribute
1671 `setID`).

```
1672 <ads:QueryResponse
1673     xmlns:ads="http://www.example.com/2010/12/Addr"
1674     timeStamp="2004-03-23T03:40:00Z">
1675   <ads:Status code="OK"/>
1676   <ads:Data remaining="20" nextOffset="10" setID="gfkjds98">
1677     first ten address cards
1678   </ads:Data>
1679 </ads:QueryResponse>
1680
1681
```

1682 In the next query the WSC A queries the next ten address card referring to the static set using the `setID` XML attribute.
1683 The **<Select>** element is not anymore used.

```
1684 <ads:Query xmlns:ads="http://www.example.com/2010/12/Addr">
1685   <ads:QueryItem count="10" offset="10" setID="gfkjds98"/>
1686 </ads:Query>
1687
1688
```

1689 In the response the next ten address cards are returned and the `setID` is still returned as always when accessing a static
1690 set.

```
1691 <ads:QueryResponse
1692     xmlns:ads="http://www.example.com/2010/12/Addr"
1693     timeStamp="2004-03-23T03:40:00Z">
1694   <ads:Status code="OK"/>
1695   <ads:Data remaining="10" nextOffset="20" setID="gfkjds98">
1696     next ten address cards
1697   </ads:Data>
1698 </ads:QueryResponse>
```

1699
1700

1701 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
1702 ten address cards.

```
1703 <ads:Query xmlns:ads="http://www.example.com/2010/12/Addr">  
1704   <ads:QueryItem count="10" offset="20" setID="gfkjds98"/>  
1705 </ads:Query>  
1706  
1707
```

1708 So the WSC A gets the last ten address card.

```
1709 <ads:QueryResponse  
1710   xmlns:ads="http://www.example.com/2010/12/Addr"  
1711   timeStamp="2004-03-23T03:40:00Z">  
1712   <ads:Status code="OK"/>  
1713   <ads:Data remaining="0" nextOffset="30" setID="gfkjds98">  
1714     ... next ten address cards ...  
1715   </ads:Data>  
1716 </ads:QueryResponse>  
1717  
1718
```

1719 Finally the WSC A deletes the static set. This deletion could have been done together with the previous request, but
1720 the WSC wanted to play safe and delete the static set only after getting all the data it wanted.

```
1721 <ads:Query xmlns:ads="http://www.example.com/2010/12/Addr">  
1722   <ads:QueryItem count="0" setID="gfkjds98" setReq="DeleteSet"/>  
1723 </ads:Query>  
1724  
1725
```

1726 And the WSP acknowledges the request.

```
1727 <ads:QueryResponse  
1728   xmlns:ads="http://www.example.com/2010/12/Addr"  
1729   timeStamp="2004-03-23T03:40:00Z">  
1730   <ads:Status code="OK"/>  
1731 </ads:QueryResponse>  
1732  
1733
```

1734 So the addition the WSC B tried to make is not visible in the static set. Either the WSP refused to accept the addition
1735 while WSC A was accessing the data or it created a temporary set for the WSC A to access and the modification by the
1736 WSC B was accepted, but not visible in the temporary static set created for WSC A. In the example above the WSP
1737 created a temporary set and so returned the same time stamp in all responses containing data from that temporary set.

1738 5. Creating Data Objects

1739 A WSC can create new data objects to a resource when a service type supports multiple objects of the same type. If
1740 there is only one object of a type, that object exists always, when a resource containing it exists. The data objects can
1741 later be modified and deleted.

1742 5.1. <Create> Element

1743 The <Create> element is used to create new data objects, not new data inside existing data objects. The content of
1744 a data object is created, deleted and modified using the <Modify>. The right resource, to which a new data object
1745 is added, is selected using security mechanism and possibly <TargetIdentity> header. The <CreateItem> element
1746 specifies the type of the new object (the objectType XML attribute) and initial content for the new object (inside the
1747 <NewData> element). The <NewData> MAY contain some local addressing element that further qualifies the object
1748 that is being created. For example, when adding an address card, service specification may specify an address card
1749 identifier that differentiates the object from other similar objects (or this identifier may be assigned automatically by
1750 the service, in which case the <ResultQuery> may come handy to discover which identifier was assigned).

```
1751 <xs:attributeGroup name="CreateItemAttributeGroup">
1752   <xs:attribute ref="dst:objectType" use="optional"/>
1753   <xs:attribute name="id" use="optional" type="xs:ID"/>
1754   <xs:attribute ref="lu:itemID" use="optional"/>
1755 </xs:attributeGroup>
```

1756 **Figure 12. XML Attributes for CreateItem**

```
1757 <xs:complexType name="CreateType">
1758   <xs:complexContent>
1759     <xs:extension base="dst:RequestType">
1760       <xs:sequence>
1761         <xs:element ref="dstref:CreateItem" minOccurs="1" maxOccurs="unbounded"/>
1762         <xs:element ref="dstref:ResultQuery" minOccurs="0" maxOccurs="unbounded"/>
1763       </xs:sequence>
1764     </xs:extension>
1765   </xs:complexContent>
1766 </xs:complexType>
1767 <xs:element name="CreateItem" type="dstref:CreateItemType"/>
1768 <xs:complexType name="CreateItemType">
1769   <xs:sequence>
1770     <xs:element ref="dstref:NewData" minOccurs="0" maxOccurs="1"/>
1771   </xs:sequence>
1772   <xs:attributeGroup ref="dst:CreateItemAttributeGroup"/>
1773 </xs:complexType>
1774 <xs:element name="NewData" type="dstref:AppDataType"/>
1775 <xs:complexType name="CreateResponseType">
1776   <xs:complexContent>
1777     <xs:extension base="dstref:DataResponseType"/>
1778   </xs:complexContent>
1779 </xs:complexType>
1780 <xs:complexType name="DataResponseType">
1781   <xs:complexContent>
1782     <xs:extension base="dst:DataResponseBaseType">
1783       <xs:sequence>
1784         <xs:element ref="dstref:ItemData" minOccurs="0" maxOccurs="unbounded"/>
1785       </xs:sequence>
1786     </xs:extension>
1787   </xs:complexContent>
1788 </xs:complexType>
```

1789 **Figure 13. Reference Model for Create**

1790 5.2. <CreateResponse> Element

1791 The <CreateResponse> element contains in addition to the mandatory <Status> element possible <ItemData>
1792 elements, which carry requested data related to the data just created. For example, returned data could include a
1793 unique ID assigned to the data object just created.

1794 5.3. Processing Rules for Creating Data Objects

1795 The common processing rules specified earlier MUST also be followed (see [Section 3](#)).

1796 5.3.1. Multiple <CreateItem> Elements

1797 One <Create> element can contain multiple <CreateItem> elements. The following rules specify how those must be
1798 supported and handled:

1799 1. A WSP MUST support one <CreateItem> element inside a <Create> and SHOULD support multiple. If a WSP
1800 supports only one <CreateItem> element inside a <Create> and the <Create> contains multiple <CreateItem>
1801 elements, the processing of the whole <Create> MUST fail and a status code indicating failure MUST be returned
1802 in the response. A more detailed status code with the value `NoMultipleAllowed` SHOULD be used in addition
1803 to the top level status code. If a WSP supports multiple <CreateItem> elements inside a <Create>, it MAY
1804 register the urn:liberty:dst:multipleCreateItems discovery option keyword.

1805 2. If the processing of a <CreateItem> fails even partly due to some reason, depending on the service and/or a
1806 WSP either the processing of the whole <Create> MUST fail or a WSP MUST try to achieve partial success.
1807 The top level status code `Failed` or `Partial` MUST be used to indicate the failure (complete or partial) and a
1808 more detailed second level status code SHOULD be used to indicate the reason for failing to completely process
1809 the failed <Create> element. Furthermore, the `ref` XML attribute of the <Status> element SHOULD carry the
1810 value of the `itemID` of the failed <CreateItem> element and in partial success cases it MUST carry this value.
1811 The modifications made based on already processed <CreateItem> elements of the <Create> MUST be rolled
1812 back in case of a complete failure. A WSP MUST NOT support multiple <CreateItem> elements inside one
1813 <Create>, if it cannot roll back and partial failure is not allowed.

1814 3. When multiple <CreateItem> elements inside one <Create> element are supported and partial success is
1815 allowed, a WSC MUST use the `itemID` XML attribute in each <CreateItem> element so that a WSP can
1816 identify the failed parts, when it is returning status information for a partial success.

1817 5.3.2. Only One Type of Data Object per <CreateItem>

1818 With one <CreateItem> element a WSC can add only one type of data objects, but the amount of object may vary.

1819 1. A WSP MUST support multiple data objects of the same type inside the <NewData> element of a <CreateItem>
1820 element, if the service can have multiple objects of that type, unless otherwise specified in a service specification.
1821 If a data object inside a <NewData> element is not of the type specified by the `objectType` XML attribute
1822 of the <CreateItem> containing the <NewData> element, the processing of that <CreateItem> MUST fail
1823 and second level status code `ObjectTypeMismatch` SHOULD be used. If the data inside a <NewData> is
1824 otherwise unacceptable to a WSP, the processing of the <CreateItem> MUST fail and second level status code
1825 `InvalidData` SHOULD be used unless some better service or object type specific status code has been defined
1826 in the service specification or in this specification. A data object might contain an <Extension> element, which
1827 has some data not specified in the service specification. A WSP might not support extensions and not accept that
1828 data. This SHOULD be indicated with the second level status code `ExtensionNotSupported`.

1829 2. If there is no `<NewData>` element inside a `<CreateItem>`, an empty data object of the type specified by the
1830 `objectType` XML attribute MUST be created unless service specification requires that a object always has
1831 some data, e.g., an identifier created by a WSC to be used to access that specific object instead of other objects
1832 of the same type. If a `<NewData>` element is required inside a `<CreateItem>` element and it is missing, the
1833 processing of that `<CreateItem>` MUST fail and second level status code `MissingNewData` should be used to
1834 indicate this.

1835 5.3.3. Handling `commonAttributes` and `leafAttributes` upon Creation

1836 The common XML attributes belonging to the XML attribute groups `commonAttributes` and `leafAttributes` are
1837 mainly supposed to be written by the WSP hosting the data service. There are some additional rules for handling these
1838 common XML attributes when data objects are created.

1839 1. When any of the `ACC`, `modifier`, `ACCTime` or `modificationTime` XML attributes is used in a resource, the
1840 WSP hosting the data service MUST keep their values up to date. When a data object is created, the `modifier`
1841 XML attribute MUST contain the `ProviderID` of the creator or have no value, and the `modificationTime`
1842 MUST define the time of the creation or have no value. The `ACC` MUST define the XML attribute collection
1843 context of the current value of a data element or have no value and the `ACCTime` MUST define the time, when
1844 the value of the `ACC` was defined or have no value.

1845 2. If the `<NewData>` contains `modifier`, `modificationTime` or `ACCTime` XML attributes for any data element,
1846 the WSP MUST ignore these and update the values based on other information than those XML attributes inside
1847 the `<NewData>` provided by the WSC. If the `ACC` XML attribute is included for any data element, the WSP
1848 MAY accept it, depending on how much it trusts the requesting service provider. The WSP MAY also accept the
1849 `id` XML attribute provided inside the `<NewData>` and some services MAY require that the `id` XML attribute
1850 MUST be provided by the requesting WSC.

1851 3. The `id` XML attribute MUST NOT be used as a global unique identifier. The value MUST be chosen so that it
1852 works only as unique identifier inside the conceptual XML document.

1853 4. When a data object is created based on a `<Create>` request, the values of the `modificationTime` XML
1854 attributes written by the WSP hosting the data service MAY be same for all elements of created object, but
1855 there is no guarantee that they will be exactly the same. When the `modificationTime` XML attribute is used
1856 in container elements, the time of a modification MUST be propagated to all ancestor elements of the modified
1857 element all the way up to the root element. So the root element has always the latest modification time.

1858 5.3.4. WSC Might Not Be Allowed to Add Certain Data or Any Data

1859 When a WSP processes a `<CreateItem>`, it MUST check, whether the resource owner (for example, the Principal)
1860 has given consent to the requestor to create the data. To be able to check WSC-specific access rights, the WSP
1861 MUST authenticate the WSC (see [[LibertySecMech](#)]). If the consent check fails for any part of the requested data, the
1862 WSP MUST NOT create data requested in the `<CreateItem>` element, even when such consent is missing only
1863 for some subelement or XML attribute. The WSP MAY try to get consent from the Principal while processing
1864 the request perhaps using an interaction service (for more information see [[LibertyInteract](#)]). The processing of
1865 a `<CreateItem>` element MUST fail, if the creation of the data object was not allowed. The second level status
1866 code `ActionNotAuthorized` MAY be used, if it is considered that the privacy of the owner of the resource is not
1867 compromised. A WSP might check the access rights at a higher level, before getting to DST processing and MAY
1868 return an ID-* Fault Message [[LibertySOAPBinding](#)] and not process the `<Create>` element at all, if the requesting
1869 WSC is not allowed to create data objects.

1870 5.3.5. WSP May Place Some Restrictions for the data It Is Hosting

- 1871 1. The schemata for different data services may have some elements for which there is not an exact upper limit
1872 on how many can exist. For practical reasons, implementations may set some limits. If a request tries to
1873 add more elements than a WSP supports, the WSP will not accept the new element(s) and the processing of
1874 the **<CreateItem>** element MUST fail. The WSP should use a second level status code `NoMoreElements` to
1875 indicate this specific case. If a WSC tries to add more data object than a WSP supports, the processing of the
1876 **<CreateItem>** element MUST fail and the second level status code `NoMoreObjects` to indicate this. If only one
1877 data object of the type specified by the `objectType` is allowed and a WSC tries to create it although it already
1878 exists, the correct second level status code is `ExistsAlready`.
- 1879 2. The schemata for different data services may not specify the length of elements and XML attributes especially
1880 in the case of strings. If a request tries to add longer values for data elements or XML attributes than a WSP
1881 supports, the WSP may not accept the data and the processing of the **<CreateItem>** element will fail. The WSP
1882 should use a second level status code `DataTooLong` to indicate this.

1883 6. Deleting Data Objects

1884 A WSC can delete existing data objects, when a service supports multiple data objects of the same type.

1885 6.1. <Delete> Element

1886 The <Delete> element is used to delete existing data objects, not data inside a data object, but whole objects including
1887 the contained data. If only the data inside an object should be deleted, a WSC must use <Modify> for it.

1888 The data objects to be deleted are referred to either using the predefined XML attribute or the objectType
1889 XML attribute and the <Select> element in the <DeleteItem> element. Concurrent updates are handled using
1890 the notChangedSince XML attribute inside the <DeleteItem> element. If the data has been modified since the time
1891 specified by the notChangedSince XML attribute, the deletion MUST NOT be done.

```

1892 <xs:complexType name="DeleteItemBaseType">
1893   <xs:attributeGroup ref="dst:selectQualif"/>
1894   <xs:attribute name="notChangedSince" use="optional" type="xs:dateTime"/>
1895   <xs:attribute name="id" use="optional" type="xs:ID"/>
1896   <xs:attribute ref="lu:itemID" use="optional"/>
1897 </xs:complexType>
1898 <xs:complexType name="DeleteResponseType">
1899   <xs:complexContent>
1900     <xs:extension base="lu:ResponseType"/>
1901   </xs:complexContent>
1902 </xs:complexType>

```

1903 **Figure 14. Utility Schema for Delete**

```

1904 <xs:complexType name="DeleteType">
1905   <xs:complexContent>
1906     <xs:extension base="dst:RequestType">
1907       <xs:sequence>
1908         <xs:element ref="dstref:DeleteItem" minOccurs="1" maxOccurs="unbounded"/>
1909       </xs:sequence>
1910     </xs:extension>
1911   </xs:complexContent>
1912 </xs:complexType>
1913 <xs:element name="DeleteItem" type="dstref:DeleteItemType"/>
1914 <xs:complexType name="DeleteItemType">
1915   <xs:complexContent>
1916     <xs:extension base="dst:DeleteItemBaseType">
1917       <xs:sequence>
1918         <xs:element ref="dstref:Select" minOccurs="0" maxOccurs="1"/>
1919       </xs:sequence>
1920     </xs:extension>
1921   </xs:complexContent>
1922 </xs:complexType>
1923 <xs:complexType name="DeleteResponseType">
1924   <xs:complexContent>
1925     <xs:extension base="lu:ResponseType"/>
1926   </xs:complexContent>
1927 </xs:complexType>

```

1928 **Figure 15. Reference Model for Delete**

1929 6.2. <DeleteResponse> Element

1930 The <DeleteResponse> element contains mainly the mandatory <Status> element. No time stamp is returned as the
1931 data does not exist after processing the request.

1932 **6.3. Processing Rules for Deletion**

1933 The common processing rules specified earlier MUST also be followed (see [Section 3](#)).

1934 **6.3.1. Supporting Multiple <DeleteItem> Elements**

1935 One <Delete> element can contain multiple <DeleteItem> elements. The following rules specify how those must be
1936 supported and handled:

1937 1. A WSP MUST support one <DeleteItem> element inside a <Delete> and SHOULD support multiple. If a WSP
1938 supports only one <DeleteItem> element inside a <Delete> and the <Delete> contains multiple <DeleteItem>
1939 elements, the processing of the whole <Delete> MUST fail and a status code indicating failure MUST be returned
1940 in the response. A more detailed status code with the value `NoMultipleAllowed` SHOULD be used in addition
1941 to the top level status code. If a WSP supports multiple <DeleteItem> elements inside a <Delete>, it MAY
1942 register the urn:liberty:dst:multipleDeleteItems discovery option keyword.

1943 2. If the processing of a <DeleteItem> fails even partly due to some reason, depending on the service and/or a WSP
1944 either the processing of the whole <Delete> MUST fail or a WSP MUST try to achieve partial success. The top
1945 level status code `Failed` or `Partial` MUST be used to indicate the failure (complete or partial) and a more
1946 detailed second level status code SHOULD be used to indicate the reason for failing to completely process the
1947 failed <Delete> element. Furthermore, the `ref` XML attribute of the <Status> element SHOULD carry the
1948 value of the `itemID` of the failed <DeleteItem> element and in partial success cases it MUST carry this value.
1949 The deletions made based on already processed <DeleteItem> elements of the <Delete> MUST be rolled back in
1950 case of a complete failure. A WSP MUST NOT support multiple <DeleteItem> elements inside one <Delete>,
1951 if it cannot roll back and partial failure is not allowed.

1952 3. When multiple <DeleteItem> elements inside one <Delete> element are supported and partial success is allowed,
1953 a WSC MUST use the `itemID` XML attribute in each <DeleteItem> element so that a WSP can identify the failed
1954 parts, when it is returning status information for a partial success.

1955 **6.3.2. Only One Type of Data Object May Be Deleted with One <DeleteItem>**

1956 With one <DeleteItem> element a WSC can delete only one type of data objects unless `predefined` XML attribute
1957 is used, but the amount of object may vary.

1958 1. All data objects matching the selection criteria given in a <DeleteItem>, either `predefined` XML attribute or
1959 `objectType` XML attribute and <Select> element, MUST be deleted. If all matching can not be deleted, the
1960 processing of that <DeleteItem> MUST fail and appropriate second level status code should be used to indicate
1961 the reason. If a <DeleteItem> fails, a WSP MUST NOT delete any data based on it.

1962 2. If there is no <Select> element inside a <DeleteItem>, all data objects of the type specified by the `objectType`
1963 XML attribute MUST be deleted. A service specification may require that <Select> element is always used, when
1964 the `predefined` XML attribute is not used.

1965 **6.3.3. Avoiding Deletion of Data if It Has Changed In-between**

1966 A WSC might want to avoid deleting data, if someone else has changed it in-between.

1967 When the `notChangedSince` XML attribute is present, the deletions specified by a `<DeleteItem>` element MUST
1968 NOT be made, if any part of the data to be deleted has changed since the time specified by the `notChangedSince`
1969 XML attribute. The second level status code `ModifiedSince` MUST be used to indicate that the deletion was not
1970 done because the data has been modified since the time specified by the `notChangedSince` XML attribute. If a WSP
1971 does not support processing of this XML attribute properly, it MUST NOT make any changes and it MUST return the
1972 second level status code `ChangeHistoryNotSupported`. If a WSP supports this `notChangedSince` XML attribute,
1973 it MUST also support the `changedSince` XML attribute of the `<QueryItem>` element and `notChangedSince` XML
1974 attribute of the `<ModifyItem>`.

1975 **6.3.4. WSC Might Not Be Allowed to Delete Certain or Any Data**

1976 When a WSP processes a `<DeleteItem>`, it MUST check, whether the resource owner (for example, the Principal)
1977 has given consent to the requestor to delete the data. To be able to check WSC-specific access rights, the WSP MUST
1978 authenticate the WSC (see [[LibertySecMech](#)]). If the consent check fails for any part of the data requested to be
1979 deleted, the WSP MUST NOT delete data requested in the `<DeleteItem>` element, even when such consent is missing
1980 only for some subelement or XML attribute. The WSP MAY try to get consent from the Principal while processing
1981 the request, for example, using an interaction service (for more information see [[LibertyInteract](#)]). The processing
1982 of a `<DeleteItem>` element MUST fail, if the deletion of a data object was not allowed. The second level status
1983 code `ActionNotAuthorized` MAY be used, if it is considered that the privacy of the owner of the resource is not
1984 compromised. A WSP might check the access rights at a higher level, before getting to DST processing and MAY
1985 return an ID-* Fault Message [[LibertySOAPBinding](#)] and not process the `<Delete>` element at all, if the requesting
1986 WSC is not allowed to delete data objects.

1987 7. Modifying Data

1988 The data objects stored by a data service can be modified. Usually the Principal can make these modifications directly
1989 at the data service using the provided user interface, but these modifications may also be made by other service
1990 providers using the **<Modify>** element. It is not possible to create or delete data objects with the **<Modify>**, just
1991 change of existing data objects.

1992 7.1. **<Modify>** Element

1993 The **<Modify>** element has two types of sub-elements.

1994 • **<ModifyItem>** elements specify which data elements of the specified resource should be modified and how.

1995 • **<ResultQuery>** elements can be included, when a WSC wants, for example, to get back data related to the
1996 modifications it just made.

1997 The `objectType` XML attribute and the **<Select>** element inside a **<ModifyItem>** element specifies the data this
1998 modification should affect. The **<Select>** element is not needed when a resource in a data service has only one data
1999 object of the type specified with the `objectType` XML attribute and the whole content of that data object is modified.
2000 If a data service supports only one `objectType`, this XML attribute may be omitted.

2001 The **<NewData>** subelement of **<ModifyItem>** defines the new values for the data addressed by the `objectType`
2002 XML attribute and the **<Select>** element. The new values, specified inside the **<NewData>** element, replace any
2003 existing selected data, if the `overrideAllowed` XML attribute of the **<ModifyItem>** element is set to `True`.

2004 If the **<NewData>** element does not exist or is empty, it means than the selected current data values should be removed.
2005 Note that whole data object can be deleted only with a separate **<Delete>** message, not with **<Modify>**. The default
2006 value for the `overrideAllowed` XML attribute is `False`, which means that the **<ModifyItem>** is only allowed to
2007 add new data to a data object, not to remove or replace existing data of a data object.

2008 The `notChangedSince` XML attribute is used to handle concurrent updates. When the `notChangedSince` XML
2009 attribute is present, a modification is allowed to be done only if the data to be modified has not changed since the time
2010 specified by the value of the `notChangedSince` XML attribute.

2011 The **<ModifyItem>** element **MUST** also have the `itemID` XML attribute, when multiple **<ModifyItem>** elements
2012 are included in one **<Modify>** element and partial failure is allowed so that failed parts can be identified.

2013 A **<Modify>** may include **<ResultQuery>** elements, if a WSC wants to get back data it is just modifying to, for
2014 example, find out the details, was all the new data accepted, or get back possible metadata a WSP might have added
2015 to the modified data.

```
2016 <xs:attributeGroup name="ModifyItemAttributeGroup">  
2017   <xs:attributeGroup ref="dst:selectQualif"/>  
2018   <xs:attribute name="notChangedSince" use="optional" type="xs:dateTime"/>  
2019   <xs:attribute name="overrideAllowed" use="optional" type="xs:boolean" default="0"/>  
2020   <xs:attribute name="id" use="optional" type="xs:ID"/>  
2021   <xs:attribute ref="lu:itemID" use="optional"/>  
2022 </xs:attributeGroup>
```

2023 **Figure 16. XML Attributes for Modify**

```
2024 <xs:complexType name="ModifyType">
2025   <xs:complexContent>
2026     <xs:extension base="dst:RequestType">
2027       <xs:sequence>
2028         <xs:element ref="dstref:ModifyItem" minOccurs="1" maxOccurs="unbounded"/>
2029         <xs:element ref="dstref:ResultQuery" minOccurs="0" maxOccurs="unbounded"/>
2030       </xs:sequence>
2031     </xs:extension>
2032   </xs:complexContent>
2033 </xs:complexType>
2034 <xs:element name="ModifyItem" type="dstref:ModifyItemType"/>
2035 <xs:complexType name="ModifyItemType">
2036   <xs:sequence>
2037     <xs:element ref="dstref:Select" minOccurs="0" maxOccurs="1"/>
2038     <xs:element ref="dstref:NewData" minOccurs="0" maxOccurs="1"/>
2039   </xs:sequence>
2040   <xs:attributeGroup ref="dst:ModifyItemAttributeGroup"/>
2041 </xs:complexType>
2042 <xs:complexType name="ModifyResponseType">
2043   <xs:complexContent>
2044     <xs:extension base="dstref:DataResponseType"/>
2045   </xs:complexContent>
2046 </xs:complexType>
```

2047 **Figure 17. Reference Model for Modify**2048 **7.2. <ModifyResponse> Element**

2049 The **<ModifyResponse>** element contains the **<Status>** element, which describes whether or not the requested
2050 modification succeeded. There is also a possible time stamp XML attribute, which provides a time value that can
2051 be used later to check whether there have been any changes since this modification, and an `itemIDRef` XML attribute
2052 to map the **<ModifyResponse>** elements to the **<Modify>** elements in the request.

2053 A **<ModifyResponse>** may also contain **<ItemData>** elements which contain data requested with **<ResultQuery>**
2054 elements. One **<ItemData>** element MUST NOT contain more data than requested with one **<ResultQuery>** element.
2055 Note that a WSP MAY return data using the **<ItemData>** element even when a WSC did not ask for it, if a WSP thinks
2056 that a WSC needs that data, e.g., to access it later on.

2057 **7.3. Processing Rules for Modifications**

2058 The common processing rules specified earlier MUST also be followed (see [Section 3](#)).

2059 **7.3.1. Multiple <ModifyItem> Elements**

2060 1. A WSP MUST support one **<ModifyItem>** element inside a **<Modify>** and SHOULD support multiple. If the
2061 **<Modify>** contains multiple **<ModifyItem>** elements and the WSP supports only one **<ModifyItem>** element
2062 inside a **<Modify>**, the processing of the whole **<Modify>** MUST fail and a status code indicating failure
2063 MUST be returned in the response. The value `NoMultipleAllowed` SHOULD be used for the second level
2064 status code. If a WSP supports multiple **<ModifyItem>** element inside a **<Modify>**, it MAY register the
2065 urn:liberty:dst:multipleModifyItem discovery option keyword.

- 2066 2. If the processing of a **<ModifyItem>** fails even partly due to some reason, depending on the service and/or a
2067 WSP either the processing of the whole **<Modify>** MUST fail or a WSP MUST try to achieve partial success.
2068 The top level status code `Failed` or `Partial` MUST be used to indicate the failure (complete or partial) and a
2069 more detailed second level status code SHOULD be used to indicate the reason for failing to completely process
2070 the failed **<Modify>** element. Furthermore, the `ref` XML attribute of the **<Status>** element SHOULD carry the
2071 value of the `itemID` of the failed **<ModifyItem>** element and in partial success cases it MUST carry this value.
2072 The modifications made based on already processed **<ModifyItem>** elements of the **<Modify>** MUST be rolled
2073 back in case of a complete failure. A WSP MUST NOT support multiple **<ModifyItem>** elements inside one
2074 **<Modify>**, if it cannot roll back and partial failure is not allowed.
- 2075 3. When multiple **<ModifyItem>** elements inside one **<Modify>** element are supported and partial success is
2076 allowed, a WSC MUST use the `itemID` XML attribute in each **<ModifyItem>** element so that a WSP can
2077 identify the failed parts, when it is returning status information for a partial success.

2078 7.3.2. What Exactly Is Modified

2079 What is modified and how depends on a number of parameters including the value of the **<Select>** element, the content
2080 of the provided **<NewData>** element, the value of the `overrideAllowed` XML attribute, and the current content of
2081 the underlying conceptual XML document.

- 2082 1. When adding new data, the **<Select>** element will point in the conceptual XML document to an element which
2083 does not exist yet. The new element is added as a result of processing the **<ModifyItem>** element. In such cases,
2084 when the ancestor elements of the new element do not exist either, they MUST be added as part of processing of
2085 the **<ModifyItem>** element so that processing could be successful.
- 2086 2. If the **<Select>** points to multiple places and there is a **<NewData>** element with new values, the processing of the
2087 **<ModifyItem>** MUST fail because it is not clear where to store the new data. If there is no **<NewData>** element
2088 and the `overrideAllowed` XML attribute is set to `True`, then the processing of **<ModifyItem>** can continue
2089 normally, because it is acceptable to delete multiple data elements at once (for example, all `AddressCards`).
- 2090 3. When the `overrideAllowed` is set to `False` or is missing, the **<NewData>** element MUST be present as new
2091 data should be added. If the **<NewData>** element is missing in this case, the processing of the **<ModifyItem>**
2092 MUST fail and the second level status code `MissingNewDataElement` SHOULD be returned in addition to top
2093 level status code.
- 2094 4. When there is the **<NewData>** element with new values and the **<Select>** points to existing information, the
2095 processing of the **<ModifyItem>** MUST fail, if the `overrideAllowed` XML attribute is not set to `True`. When
2096 the `overrideAllowed` XML attribute does not exist or is set to `False`, the new data in the **<NewData>** element
2097 can only be accepted in two cases: either there is no existing element to which the **<Select>** points, or there can be
2098 multiple data elements of the same type. This means that, if the **<Select>** points to an existing container element,
2099 which has a subelement, and only one such container element can exist, the **<ModifyItem>** MUST fail, even if
2100 the only subelement the container element has inside the **<NewData>** does not yet exist in the conceptual XML
2101 document. The second level status code `ExistsAlready` SHOULD be used to indicate in details the reason for
2102 the failure in addition to the top level status code. The lack of those other sub-elements inside the **<NewData>**
2103 means that they should be removed, which is only possible when `overrideAllowed` XML attribute equals to
2104 `True`.
- 2105 5. When there can be multiple elements of the same type, the addition of a new element MUST fail, if there exists
2106 already an element of same type have the same value of the distinguishing part. In the case of a personal profile
2107 service, adding a new **<AddressCard>** element MUST fail, if there already exists an **<AddressCard>** element
2108 which has an `id` XML attribute of the same value as the provided new **<AddressCard>** element. The second
2109 level status code `ExistsAlready` SHOULD also be used to indicate the detailed reason for failure.

- 2110 6. When all or some of the data inside the `<NewData>` element is not supported by the WSP, or the provided data is
2111 not valid, the processing of the whole `<ModifyItem>` SHOULD fail and second level status code `InvalidData`
2112 SHOULD be returned in the response.
- 2113 7. When the `<ModifyItem>` element tries to extend the service either by pointing to a new data type behind an
2114 `<Extension>` element with the `<Select>` element, or having new sub-elements under an `<Extension>` element
2115 inside the `<NewData>` element and the WSP does not support extension in general or for the requesting party, it
2116 SHOULD be indicated in the response message with the second level status code `ExtensionNotSupported`.
- 2117 8. When the WSP supports extensions, but does not accept the content of the `<Select>` or `<NewData>`, then second
2118 level status codes `InvalidSelect` and `InvalidData` SHOULD be used as already described.

2119 7.3.3. Handling `commonAttributes` and `leafAttributes` in `Modify`

2120 The common XML attributes belonging to the XML attribute groups `commonAttributes` and `leafAttributes` are
2121 mainly supposed to be written by the WSP hosting the data service. There are some additional rules for handling
2122 these common XML attributes in case of modifications.

- 2123 1. When any of the `ACC`, `modifier`, `ACCTime` or `modificationTime` XML attributes is used in a resource, the
2124 WSP hosting the data service MUST keep their values up to date. When data is modified, the `modifier` MUST
2125 contain the `ProviderID` of the modifier or have no value, and the `modificationTime` MUST define the time of
2126 the modification or have no value. The `ACC` MUST define the XML attribute collection context of the current
2127 value of a data element or have no value and the `ACCTime` MUST define the time, when the current value of the
2128 `ACC` was defined or have no value.
- 2129 2. If the `<NewData>` contains `modifier`, `modificationTime` or `ACCTime` XML attributes for any data element,
2130 the WSP MUST ignore these and update the values based on other information than those XML attributes inside
2131 the `<NewData>` provided by the WSC. If the `ACC` XML attribute is included for any data element, the WSP
2132 MAY accept it, depending on how much it trusts the requesting service provider. The WSP MAY also accept the
2133 `id` XML attribute provided inside the `<NewData>` and some services MAY require that the `id` XML attribute
2134 MUST be provided by the requesting service provider.
- 2135 3. The `id` XML attribute MUST NOT be used as a global unique identifier. The value MUST be chosen so that
2136 it works only as unique identifier inside the conceptual XML document, and the value of the `id` XML attribute
2137 SHOULD be kept the same even if the element is otherwise modified. A WSP MAY not even allow changing the
2138 value of the `id` XML attribute or any other XML attribute used to distinguish elements with the same name from
2139 each other.
- 2140 4. When data is modified based on the `<Modify>` request, the values of the `modificationTime` XML attributes
2141 written by the WSP hosting the data service MAY be same for all inserted and updated elements, but there is
2142 no guarantee that they will be exactly the same. When the `modificationTime` XML attribute is used by a
2143 data service, the WSP MUST keep it up to date to indicate the time of the latest modification of an element
2144 and update it, when ever a modification is done either using the `<Modify>` request or some other way. When
2145 the `modificationTime` XML attribute is used in container elements, the time of a modification MUST be
2146 propagated to all ancestor elements of the modified element all the way up to the root element.

2147 7.3.4. Accounting for Concurrent Updates

2148 Accounting for concurrent updates is handled using the `notChangedSince` XML attribute inside the `<ModifyItem>`
2149 element.

- 2150 1. When the `notChangedSince` XML attribute is present, the modifications specified by the `<ModifyItem>`
2151 element MUST NOT be made, if any part of the data to be modified has changed since the time specified by
2152 the `notChangedSince` XML attribute.

2153 2. The second level status code `ModifiedSince` MUST be used to indicate that the modification was not done
2154 because the data has been modified since the time specified by the `notChangedSince` XML attribute. If a WSP
2155 does not support processing of this XML attribute properly, it MUST NOT make any changes and it MUST return
2156 the second level status code `ChangeHistoryNotSupported`. If a WSP supports this `notChangedSince` XML
2157 attribute, it MUST also support the `changedSince` XML attribute of the `<QueryItem>` element.

2158 7.3.5. WSC Might Not Be Allowed to Make Only Certain or Any Modifications

2159 When a WSP processes the `<ModifyItem>`, it MUST check, whether the resource owner (for example, the Principal)
2160 has given consent to the requestor to modify the data. To be able to check WSC-specific access rights, the WSP
2161 MUST authenticate the WSC (see [[LibertySecMech](#)]). If the consent check fails for any part of the requested data,
2162 the WSP MUST NOT make the modifications requested in the `<ModifyItem>` element, even when such consent
2163 is missing only for some subelement or XML attribute. The WSP MAY try to get consent from the Principal
2164 while processing the request perhaps using an interaction service (for more information see [[LibertyInteract](#)]). The
2165 processing of the `<ModifyItem>` element MUST fail, if the modification was not allowed. The second level status
2166 code `ActionNotAuthorized` MAY be used, if it is considered that the privacy of the owner of the resource is not
2167 compromised. A WSP might check the access rights at a higher level, before getting to DST processing and MAY
2168 return an ID-* Fault Message [[LibertySOAPBinding](#)] and not process the `<Modify>` element at all, if the requesting
2169 WSC is not allowed to modify the data.</para>

2170 7.3.6. WSP May Impose Some Restrictions for the Data It Is Hosting

2171 1. The schemata for different data services may have some elements for which there is not an exact upper limit
2172 on how many can exist. For practical reasons, implementations may set some limits. If a request tries to add
2173 more elements than a WSP supports, the WSP will not accept the new element(s) and the processing of the
2174 `<ModifyItem>` element MUST fail. The WSP should use a second level status code `NoMoreElements` to
2175 indicate this specific case.

2176 2. The schemata for different data services may not specify the length of elements and XML attributes especially
2177 in the case of strings. The WSP may also have limitations of this kind. If a request tries to add longer data
2178 elements or XML attributes than a WSP supports, the WSP may not accept the data and the processing of the
2179 `<ModifyItem>` element will fail. The WSP should use a second level status code `DataTooLong` to indicate this
2180 specific case.

2181 7.4. Examples of Modifications

2182 This example adds a home address to the personal profile of a Principal:

```
2183 <hp:Modify xmlns:hp="urn:liberty:hp:2005-07">
2184   <hp:ModifyItem>
2185     <hp:Select>/hp:HP/hp:AddressCard</hp:Select>
2186     <hp:NewData>
2187       <hp:AddressCard id='98123'>
2188         <hp:AddressType>
2189           urn:liberty:hp:addrType:home
2190         </hp:AddressType>
2191         <hp:Address>
2192           <hp:PostalAddress>
2193             c/o Carolyn Lewis$2378 Madrona Beach Way North
2194           </hp:PostalAddress>
2195           <hp:PostalCode>98503-2341</hp:PostalCode>
2196           <hp:L>Olympia</hp:L>
2197           <hp:ST>wa</hp:ST>
2198           <hp:C>us</hp:C>
2199         </hp:Address>
2200       </hp:AddressCard>
2201     </hp:NewData>
2202   </hp:ModifyItem>
2203 </hp:Modify>
```

2204
2205

2206 The following example replaces the current home address with a new home address in the personal profile of a
2207 Principal. Please note that this request will fail if there are two or more home addresses in the profile, because it
2208 is not clear in this request, which of those addressed should be replaced by this address. In such a case the `id` XML
2209 attribute should be used to explicitly point which of the addresses should be changed.

```
2210 <hp:Modify xmlns:hp="urn:liberty:hp:2005-07">
2211   <hp:ModifyItem overrideAllowed="True">
2212     <hp:Select>
2213       /hp:HP/hp:AddressCard
2214       [hp:AddressType='urn:liberty:id-sis-hp:addrType:home' ]
2215     </hp:Select>
2216     <hp:NewData>
2217       <hp:AddressCard id="98123">
2218         <hp:AddressType>
2219           urn:liberty:id-sis-hp:addrType:home
2220         </hp:AddressType>
2221         <hp:Address>
2222           <hp:PostalAddress>
2223             c/o Carolyn Lewis$2378 Madrona Beach Way
2224           </hp:PostalAddress>
2225           <hp:PostalCode>98503-2342</hp:PostalCode>
2226           <hp:L>Olympia</hp:L>
2227           <hp:ST>wa</hp:ST>
2228           <hp:C>us</hp:C>
2229         </hp:Address>
2230       </hp:AddressCard>
2231     </hp:NewData>
2232   </hp:ModifyItem>
2233 </hp:Modify>
2234
2235
```

2236 This example replaces the current address identified by an `id` of '98123' with a new home address, if that address has
2237 not been modified since 12:40:01 21th January 2003 UTC.

```
2238 <hp:Modify xmlns:hp="urn:liberty:hp:2005-07">
2239   <hp:ModifyItem
2240     notChangedSince="2003-01-21T12:40:01Z "
2241     overrideAllowed="True">
2242     <hp:Select>/hp:HP/hp:AddressCard[@hp:id='98123']</hp:Select>
2243     <hp:NewData>
2244       <hp:AddressCard id="98123">
2245         <hp:AddressType>
2246           urn:liberty:id-sis-hp:addrType:home
2247         </hp:AddressType>
2248         <hp:Address>
2249           <hp:PostalAddress>
2250             c/o Carolyn Lewis$2378 Madrona Beach Way South
2251           </hp:PostalAddress>
2252           <hp:PostalCode>98503-2398</hp:PostalCode>
2253           <hp:L>Olympia</hp:L>
2254           <hp:ST>wa</hp:ST>
2255           <hp:C>us</hp:C>
2256         </hp:Address>
2257       </hp:AddressCard>
2258     </hp:NewData>
2259   </hp:ModifyItem>
2260 </hp:Modify>
2261
2262
```

2263 The following example adds another home address to the personal profile of a Principal. An id is provided for the
 2264 new address.

```

2265 <hp:Modify xmlns:hp="urn:liberty:hp:2005-07">
2266 <hp:ModifyItem>
2267 <hp:Select>
2268 /hp:HP/hp:AddressCard
2269 [hp:AddressType='urn:liberty:id-sis-hp:addrType:home']
2270 </hp:Select>
2271 <hp:NewData>
2272 <hp:AddressCard id="12398">
2273 <hp:AddressType>
2274 urn:liberty:id-sis-hp:addrType:home
2275 </hp:AddressType>
2276 <hp:Address>
2277 <hp:PostalAddress>1234 Beach Way</hp:PostalAddress>
2278 <hp:PostalCode>98765-1234</hp:PostalCode>
2279 <hp:L>Olympia</hp:L>
2280 <hp:ST>wa</hp:ST>
2281 <hp:C>us</hp:C>
2282 </hp:Address>
2283 </hp:AddressCard>
2284 </hp:NewData>
2285 </hp:ModifyItem>
2286 </hp:Modify>
2287
2288
  
```

2289 The following example removes all current home addresses from the personal profile of a Principal:

```

2290 <hp:Modify xmlns:hp="urn:liberty:hp:2005-07">
2291 <hp:ModifyItem overrideAllowed="True">
2292 <hp:Select>
2293 /hp:HP/hp:AddressCard
2294 [hp:AddressType='urn:liberty:id-sis-hp:addrType:home']
2295 </hp:Select>
2296 </hp:ModifyItem>
2297 </hp:Modify>
2298
2299
  
```

2300 The response for a valid **<Modify>** is as follows:

```

2301 <hp:ModifyResponse
2302 xmlns:hp="urn:liberty:hp:2005-07"
2303 timeStamp="2003-03-23T03:40:00Z">
2304 <hp:StatusCode="OK"/>
2305 </hp:ModifyResponse>
2306
2307
  
```

2308 **8. WSF-1.1 Compatibility**

2309 This version (2.1) of DST was designed to work well with ID-WSF 2.0 specification family. Since it is a major version
2310 upgrade, a decision was made to break the ID-WSF 1.1 compatibility, mainly by elimination of the <**ResourceIDs**>
2311 (see also [Section 3.6](#)).

2312 However, the two ID-WSF versions remain broadly compatible. [[LibertyDisco](#)], Section 10 "ID-WSF 1.x Resource
2313 Offering conveyed in an EPR" provides a method for constructing <**ResourceID**>s from credentials as well as
2314 making credentials and end points given knowledge of the `ResourceID`. The frame work version header, see
2315 [[LibertySOAPBinding](#)] allows simultaneous support to be implemented at run time.

2316 **9. Actions**

2317 When SOAP action names are need, they **SHOULD** be formed by appending to *service type* one of the Request names,
2318 i.e., Create, Delete, Query, Modify, etc.

2319 **Example**

2320 urn:liberty:id-sis-dap:2005-10:dst-2.1:Query
2321
2322

2323 10. Checklist for Service Specifications

2324 The following is a checklist of issues which should be addressed by individual service type specifications. Such
2325 specifications should always state which optional features of the DST they support, in addition to defining more
2326 general things such as discovery option keywords and the `SelectType` XML type used by the service type. A service
2327 specification should complete this list with the specific values and statements required by the specification.

2328 For optional features, the language specified by [RFC2119] MUST be used to define whether these features are
2329 available for implementations and deployments. For example, specifying that a feature 'MAY' be implemented by
2330 a WSP means that WSPs may or may not support the feature, and that WSCs should be ready to handle both cases.

2331 Default feature support policy is that all features, unless expressly waived by service specification, MUST be
2332 supported, but each feature MAY be disabled administratively or by configuration in a deployment (e.g., to provide
2333 read or write only service).

2334 1. Specify service type. Specify namespaces for the service if different from service type.

2335 2. Provide definition service schema including the methods as elements based on DST types. A service need not
2336 define every possible method supported by DST and it may define additional methods supported by service
2337 specific schema. The service may also rename some of the methods. If it does rename, it MUST state which DST
2338 method corresponds to the renamed method. There can be several service methods that map to one DST method.

2339 3. Enumerate object types

2340 4. Describe `AppDataType` and its contents. The description can come in form of XML schema, or the description
2341 can simply describe the contents of the string that is to appear in elements derived from `AppDataType`, i.e.,
2342 `<NewData>`, `<Data>`, and `<ItemData>`. The data description may make allowance for different object types.

2343 5. Describe `SelectType` and how it applies to various types of objects. If selects can not be described as a string,
2344 e.g., XPath can, the service may want to redefine the type using `xs:redefine`.

2345 It is possible that different query language or dialect is applied depending on which object type is being queried.
2346 If so, the service specification MUST resolve how to represent the different languages using one `SelectType`

2347 6. Describe `TestOpType`, considering how to test all object types supported by the specification. It is possible that
2348 different test language or dialect is applied depending on which object type is being tested. If so, the service
2349 specification MUST resolve how to represent the different languages using one `TestOpType`

2350 7. Describe `SortType`.

2351 8. Enumerate Methods and state the required level of support. The default set of methods is `<Create>`, `<Query>`,
2352 `<Modify>`, and `<Delete>`.

2353 Default method support policy is as follows

2354 a. All methods MUST be supported, but each method MAY be disabled administratively or by configuration
2355 in a deployment (e.g. to provide read only or write only service).

2356 b. If queries are disabled or access control makes it implausible that they succeed, discovery option keyword
2357 `urn:liberty:dst:noQuery` MUST be registered.

2358 c. If creates are disabled or access control makes it implausible that they succeed, discovery option keyword
2359 `urn:liberty:dst:noCreate` MUST be registered.

2360 d. If deletes are disabled or access control makes it implausible that they succeed, discovery option keyword
2361 `urn:liberty:dst:noDelete` MUST be registered.

2362 e. If modifies are disabled or access control makes it implausible that they succeed, discovery option keyword
2363 urn:liberty:dst:noModify MUST be registered.

2364 9. The discovery option keywords (see [[LibertyDisco](#)]) can either be listed with semantics here, or via a reference to
2365 the correct chapter in the specification. Please note that the DST defines the following discovery option keywords
2366 and the service specification must list which of these the service may use:

2367 urn:liberty:dst:allPaths
2368 urn:liberty:dst:can:extend
2369 urn:liberty:dst:changeHistorySupported
2370 urn:liberty:dst:contingentQueryItems
2371 urn:liberty:dst:extend
2372 urn:liberty:dst:fullXPath
2373 urn:liberty:dst:multipleCreateItems
2374 urn:liberty:dst:multipleDeleteItems
2375 urn:liberty:dst:multipleModifyItem
2376 urn:liberty:dst:multipleQueryItems
2377 urn:liberty:dst:multipleResources
2378 urn:liberty:dst:noQuery
2379 urn:liberty:dst:noCreate
2380 urn:liberty:dst:noDelete
2381 urn:liberty:dst:noModify
2382 urn:liberty:dst:noPagination
2383 urn:liberty:dst:noSorting
2384 urn:liberty:dst:noStatic
2385
2386

2387 10. Element uniqueness. State here how elements with the same name are distinguished from each other. For
2388 example, the id XML attribute is used for <AddressCard> and <MsgContact> elements, xml:lang and
2389 script XML attributes are used for localized elements, etc. Element uniqueness MUST consider different
2390 object types.

2391 11. Extension support. State whether extension is supported and if so, describe this support. A reference to the
2392 specification chapter defining this can be given. For example, "New elements and discovery option keywords
2393 MAY be defined, see chapter Y.X for more details."

2394 Extensions support should discuss both data extension and protocol extension, including <Extension> elements
2395 request and response messages.

2396 The default policy for protocol extension is that mutually consenting WSC and WSP MAY use extension points
2397 for implementation dependent purposes. Extension points that can be thus used are

2398 a. XML any extension points contained in <Extension> elements that are present in various protocol messages,
2399 provided that the extension elements are namespace qualified.

2400 b. If SelectType, TestOpType, or SortType is designated as unused by the service specification, then it
2401 MAY be used for extension, provided that the extension data is

2402 a. in URI format and use an assigned domain name as a component of the URI to ensure that extensions
2403 do no collide with each other.

2404 b. Namespace qualified XML document

2405 12. Statement of optionality of query features (and their manifestation on discovery option keywords, see above):

2406 a. Support testing

2407 b. Support <ResultQuery>

-
- 2408 c. Support sorting
 - 2409 d. Support pagination of results
 - 2410 e. Support static sets in pagination
 - 2411 f. Support multiple **<Query>** elements
 - 2412 g. Support multiple **<QueryItem>** elements
 - 2413 h. Support multiple **<TestItem>** elements
 - 2414 i. Support `changedSince` (and which formats) in **<ResultQuery>** and **<QueryItem>**
 - 2415 j. Support `includeCommonAttributes`
 - 2416 13. Statement of optionality of create features (and their manifestation on discovery option keywords, see above):
 - 2417 a. Support multiple **<Create>** elements
 - 2418 14. Statement of optionality of delete features (and their manifestation on discovery option keywords, see above):
 - 2419 a. Support multiple **<Delete>** elements
 - 2420 15. Statement of optionality of modify features (and their manifestation on discovery option keywords, see above):
 - 2421 a. Support multiple **<Modify>** elements
 - 2422 b. Support multiple **<ModifyItem>** elements
 - 2423 c. Support partial success. If multiple **<ModifyItem>** elements are supported, is partial success supported or
 - 2424 are only atomic modifications allowed?
 - 2425 d. Support `notChangedSince`

2426 11. Schemata

2427 11.1. DST Reference Model Schema

2428 The formal schema for the reference model follows.

```
2429 <?xml version="1.0" encoding="UTF-8"?>
2430 <xs:schema
2431     targetNamespace="urn:liberty:dst:2006-08:ref"
2432     xmlns:dstref="urn:liberty:dst:2006-08:ref"
2433     xmlns:dst="urn:liberty:dst:2006-08"
2434     xmlns:lu="urn:liberty:util:2006-08"
2435     xmlns:xs="http://www.w3.org/2001/XMLSchema"
2436     elementFormDefault="qualified"
2437     attributeFormDefault="unqualified">
2438   <xs:import namespace="urn:liberty:dst:2006-08"
2439     schemaLocation="liberty-idwsf-dst-v2.1.xsd"/>
2440   <xs:import namespace="urn:liberty:util:2006-08"
2441     schemaLocation="liberty-idwsf-utility-v2.0.xsd"/>
2442   <!--sec(methods)-->
2443   <xs:element name="Create" type="dstref:CreateType"/>
2444   <xs:element name="CreateResponse" type="dstref:CreateResponseType"/>
2445   <xs:element name="Query" type="dstref:QueryType"/>
2446   <xs:element name="QueryResponse" type="dstref:QueryResponseType"/>
2447   <xs:element name="Modify" type="dstref:ModifyType"/>
2448   <xs:element name="ModifyResponse" type="dstref:ModifyResponseType"/>
2449   <xs:element name="Delete" type="dstref:DeleteType"/>
2450   <xs:element name="DeleteResponse" type="dstref:DeleteResponseType"/>
2451   <!--endsec(methods)-->
2452   <!--sec(redefs)-->
2453   <xs:complexType name="SelectType">
2454     <xs:simpleContent>
2455       <xs:extension base="xs:string"/>
2456     </xs:simpleContent>
2457   </xs:complexType>
2458   <xs:complexType name="TestOpType">
2459     <xs:simpleContent>
2460       <xs:extension base="xs:string"/>
2461     </xs:simpleContent>
2462   </xs:complexType>
2463   <xs:complexType name="SortType">
2464     <xs:simpleContent>
2465       <xs:extension base="xs:string"/>
2466     </xs:simpleContent>
2467   </xs:complexType>
2468   <xs:complexType name="AppDataType">
2469     <xs:simpleContent>
2470       <xs:extension base="xs:string"/>
2471     </xs:simpleContent>
2472   </xs:complexType>
2473   <!--endsec(redefs)-->
2474   <!--sec(create)-->
2475   <xs:complexType name="CreateType">
2476     <xs:complexContent>
2477       <xs:extension base="dst:RequestType">
2478         <xs:sequence>
2479           <xs:element ref="dstref:CreateItem" minOccurs="1" maxOccurs="unbounded"/>
2480           <xs:element ref="dstref:ResultQuery" minOccurs="0" maxOccurs="unbounded"/>
2481         </xs:sequence>
2482       </xs:extension>
2483     </xs:complexContent>
2484   </xs:complexType>
2485   <xs:element name="CreateItem" type="dstref:CreateItemType"/>
2486   <xs:complexType name="CreateItemType">
2487     <xs:sequence>
2488       <xs:element ref="dstref:NewData" minOccurs="0" maxOccurs="1"/>
```

```

2489     </xs:sequence>
2490     <xs:attributeGroup ref="dst:CreateItemAttributeGroup" />
2491 </xs:complexType>
2492 <xs:element name="NewData" type="dstref:AppDataType" />
2493 <xs:complexType name="CreateResponseType">
2494     <xs:complexContent>
2495         <xs:extension base="dstref:DataResponseType" />
2496     </xs:complexContent>
2497 </xs:complexType>
2498 <xs:complexType name="DataResponseType">
2499     <xs:complexContent>
2500         <xs:extension base="dst:DataResponseBaseType">
2501             <xs:sequence>
2502                 <xs:element ref="dstref:ItemData" minOccurs="0" maxOccurs="unbounded" />
2503             </xs:sequence>
2504         </xs:extension>
2505     </xs:complexContent>
2506 </xs:complexType>
2507 <!--endsec(create)-->
2508 <!--sec(query)-->
2509 <xs:complexType name="QueryType">
2510     <xs:complexContent>
2511         <xs:extension base="dst:RequestType">
2512             <xs:sequence>
2513                 <xs:element ref="dstref:TestItem" minOccurs="0" maxOccurs="unbounded" />
2514                 <xs:element ref="dstref:QueryItem" minOccurs="0" maxOccurs="unbounded" />
2515             </xs:sequence>
2516         </xs:extension>
2517     </xs:complexContent>
2518 </xs:complexType>
2519 <xs:element name="TestItem" type="dstref:TestItemType" />
2520 <xs:complexType name="TestItemType">
2521     <xs:complexContent>
2522         <xs:extension base="dst:TestItemBaseType">
2523             <xs:sequence>
2524                 <xs:element name="TestOp" minOccurs="0" maxOccurs="1" type="dstref:TestOpType" />
2525             </xs:sequence>
2526         </xs:extension>
2527     </xs:complexContent>
2528 </xs:complexType>
2529 <xs:element name="QueryItem" type="dstref:QueryItemType" />
2530 <xs:complexType name="QueryItemType">
2531     <xs:complexContent>
2532         <xs:extension base="dstref:ResultQueryType">
2533             <xs:attributeGroup ref="dst:PaginationAttributeGroup" />
2534         </xs:extension>
2535     </xs:complexContent>
2536 </xs:complexType>
2537 <!--endsec(query)-->
2538 <!--sec(queryresp)-->
2539 <xs:complexType name="QueryResponseType">
2540     <xs:complexContent>
2541         <xs:extension base="dst:DataResponseBaseType">
2542             <xs:sequence>
2543                 <xs:element ref="dst:TestResult" minOccurs="0" maxOccurs="unbounded" />
2544                 <xs:element ref="dstref:Data" minOccurs="0" maxOccurs="unbounded" />
2545             </xs:sequence>
2546         </xs:extension>
2547     </xs:complexContent>
2548 </xs:complexType>
2549 <xs:element name="Data" type="dstref:DataType" />
2550 <xs:complexType name="DataType">
2551     <xs:complexContent>
2552         <xs:extension base="dstref:ItemDataType">
2553             <xs:attributeGroup ref="dst:PaginationResponseAttributeGroup" />
2554         </xs:extension>
2555     </xs:complexContent>

```

```

2556 </xs:complexType>
2557 <!--endsec(queryresp)-->
2558 <!--sec(mod)-->
2559 <xs:complexType name="ModifyType">
2560 <xs:complexContent>
2561 <xs:extension base="dst:RequestType">
2562 <xs:sequence>
2563 <xs:element ref="dstref:ModifyItem" minOccurs="1" maxOccurs="unbounded"/>
2564 <xs:element ref="dstref:ResultQuery" minOccurs="0" maxOccurs="unbounded"/>
2565 </xs:sequence>
2566 </xs:extension>
2567 </xs:complexContent>
2568 </xs:complexType>
2569 <xs:element name="ModifyItem" type="dstref:ModifyItemType"/>
2570 <xs:complexType name="ModifyItemType">
2571 <xs:sequence>
2572 <xs:element ref="dstref:Select" minOccurs="0" maxOccurs="1"/>
2573 <xs:element ref="dstref:NewData" minOccurs="0" maxOccurs="1"/>
2574 </xs:sequence>
2575 <xs:attributeGroup ref="dst:ModifyItemAttributeGroup"/>
2576 </xs:complexType>
2577 <xs:complexType name="ModifyResponseType">
2578 <xs:complexContent>
2579 <xs:extension base="dstref:DataResponseType"/>
2580 </xs:complexContent>
2581 </xs:complexType>
2582 <!--endsec(mod)-->
2583 <!--sec(del)-->
2584 <xs:complexType name="DeleteType">
2585 <xs:complexContent>
2586 <xs:extension base="dst:RequestType">
2587 <xs:sequence>
2588 <xs:element ref="dstref:DeleteItem" minOccurs="1" maxOccurs="unbounded"/>
2589 </xs:sequence>
2590 </xs:extension>
2591 </xs:complexContent>
2592 </xs:complexType>
2593 <xs:element name="DeleteItem" type="dstref:DeleteItemType"/>
2594 <xs:complexType name="DeleteItemType">
2595 <xs:complexContent>
2596 <xs:extension base="dst>DeleteItemBaseType">
2597 <xs:sequence>
2598 <xs:element ref="dstref:Select" minOccurs="0" maxOccurs="1"/>
2599 </xs:sequence>
2600 </xs:extension>
2601 </xs:complexContent>
2602 </xs:complexType>
2603 <xs:complexType name="DeleteResponseType">
2604 <xs:complexContent>
2605 <xs:extension base="lu:ResponseType"/>
2606 </xs:complexContent>
2607 </xs:complexType>
2608 <!--endsec(del)-->
2609 <!--sec(resqry)-->
2610 <xs:element name="Select" type="dstref:SelectType"/>
2611 <xs:element name="ResultQuery" type="dstref:ResultQueryType"/>
2612 <xs:complexType name="ResultQueryType">
2613 <xs:complexContent>
2614 <xs:extension base="dst:ResultQueryBaseType">
2615 <xs:sequence>
2616 <xs:element ref="dstref:Select" minOccurs="0" maxOccurs="1"/>
2617 <xs:element name="Sort" minOccurs="0" maxOccurs="1" type="dstref:SortType"/>
2618 </xs:sequence>
2619 </xs:extension>
2620 </xs:complexContent>
2621 </xs:complexType>
2622 <xs:element name="ItemData" type="dstref:ItemDataType"/>
    
```

```
2623 <xs:complexType name="ItemDataType">
2624 <xs:complexContent>
2625 <xs:extension base="dstref:AppDataType">
2626 <xs:attributeGroup ref="dst:ItemDataAttributeGroup" />
2627 </xs:extension>
2628 </xs:complexContent>
2629 </xs:complexType>
2630 <!--endsec(resqry)-->
2631 </xs:schema>
2632
2633
```

2634 11.2. DST Utility Schema

2635 The formal utility schema follows.

```
2636 <?xml version="1.0" encoding="UTF-8"?>
2637 <xs:schema
2638   targetNamespace="urn:liberty:dst:2006-08"
2639   xmlns:dst="urn:liberty:dst:2006-08"
2640   xmlns:lu="urn:liberty:util:2006-08"
2641   xmlns:xml="http://www.w3.org/XML/1998/namespace"
2642   xmlns:xs="http://www.w3.org/2001/XMLSchema"
2643   elementFormDefault="qualified"
2644   attributeFormDefault="unqualified">
2645 <xs:import namespace="urn:liberty:util:2006-08"
2646   schemaLocation="liberty-idwsf-utility-v2.0.xsd"/>
2647 <xs:import namespace="http://www.w3.org/XML/1998/namespace"
2648   schemaLocation="http://www.w3.org/2001/xml.xsd"/>
2649 <!--sec(ca)-->
2650 <xs:attribute name="id" type="lu:IDType"/>
2651 <xs:attribute name="modificationTime" type="xs:dateTime"/>
2652 <xs:attributeGroup name="commonAttributes">
2653 <xs:attribute ref="dst:id" use="optional"/>
2654 <xs:attribute ref="dst:modificationTime" use="optional"/>
2655 </xs:attributeGroup>
2656 <xs:attribute name="ACC" type="xs:anyURI"/>
2657 <xs:attribute name="ACCTime" type="xs:dateTime"/>
2658 <xs:attribute name="modifier" type="xs:string"/>
2659 <xs:attributeGroup name="leafAttributes">
2660 <xs:attributeGroup ref="dst:commonAttributes"/>
2661 <xs:attribute ref="dst:ACC" use="optional"/>
2662 <xs:attribute ref="dst:ACCTime" use="optional"/>
2663 <xs:attribute ref="dst:modifier" use="optional"/>
2664 </xs:attributeGroup>
2665 <xs:attribute name="script" type="xs:anyURI"/>
2666 <xs:attributeGroup name="localizedLeafAttributes">
2667 <xs:attributeGroup ref="dst:leafAttributes"/>
2668 <xs:attribute ref="xml:lang" use="required"/>
2669 <xs:attribute ref="dst:script" use="optional"/>
2670 </xs:attributeGroup>
2671 <xs:attribute name="refreshOnOrAfter" type="xs:dateTime"/>
2672 <xs:attribute name="destroyOnOrAfter" type="xs:dateTime"/>
2673 <!--endsec(ca)-->
2674 <!--sec(ct)-->
2675 <xs:complexType name="DSTLocalizedString">
2676 <xs:simpleContent>
2677 <xs:extension base="xs:string">
2678 <xs:attributeGroup ref="dst:localizedLeafAttributes"/>
2679 </xs:extension>
2680 </xs:simpleContent>
2681 </xs:complexType>
2682 <xs:complexType name="DSTString">
2683 <xs:simpleContent>
2684 <xs:extension base="xs:string">
```

```

2685     <xs:attributeGroup ref="dst:leafAttributes"/>
2686   </xs:extension>
2687 </xs:simpleContent>
2688 </xs:complexType>
2689 <xs:complexType name="DSTInteger">
2690   <xs:simpleContent>
2691     <xs:extension base="xs:integer">
2692       <xs:attributeGroup ref="dst:leafAttributes"/>
2693     </xs:extension>
2694   </xs:simpleContent>
2695 </xs:complexType>
2696 <xs:complexType name="DSTURI">
2697   <xs:simpleContent>
2698     <xs:extension base="xs:anyURI">
2699       <xs:attributeGroup ref="dst:leafAttributes"/>
2700     </xs:extension>
2701   </xs:simpleContent>
2702 </xs:complexType>
2703 <xs:complexType name="DSTDate">
2704   <xs:simpleContent>
2705     <xs:extension base="xs:date">
2706       <xs:attributeGroup ref="dst:leafAttributes"/>
2707     </xs:extension>
2708   </xs:simpleContent>
2709 </xs:complexType>
2710 <xs:complexType name="DSTMonthDay">
2711   <xs:simpleContent>
2712     <xs:extension base="xs:gMonthDay">
2713       <xs:attributeGroup ref="dst:leafAttributes"/>
2714     </xs:extension>
2715   </xs:simpleContent>
2716 </xs:complexType>
2717 <!--endsec(ct)-->
2718 <!--sec(msgintf)-->
2719 <xs:complexType name="RequestType">
2720   <xs:sequence>
2721     <xs:element ref="lu:Extension" minOccurs="0" maxOccurs="unbounded"/>
2722   </xs:sequence>
2723   <xs:attribute ref="lu:itemID" use="optional"/>
2724   <xs:anyAttribute namespace="##other" processContents="lax"/>
2725 </xs:complexType>
2726 <xs:complexType name="DataResponseBaseType">
2727   <xs:complexContent>
2728     <xs:extension base="lu:ResponseType">
2729       <xs:attribute name="timeStamp" use="optional" type="xs:dateTime"/>
2730     </xs:extension>
2731   </xs:complexContent>
2732 </xs:complexType>
2733 <!--endsec(msgintf)-->
2734 <!--sec(select)-->
2735 <xs:element name="ChangeFormat">
2736   <xs:simpleType>
2737     <xs:restriction base="xs:string">
2738       <xs:enumeration value="ChangedElements"/>
2739       <xs:enumeration value="CurrentElements"/>
2740     </xs:restriction>
2741   </xs:simpleType>
2742 </xs:element>
2743 <xs:attribute name="changeFormat">
2744   <xs:simpleType>
2745     <xs:restriction base="xs:string">
2746       <xs:enumeration value="ChangedElements"/>
2747       <xs:enumeration value="CurrentElements"/>
2748       <xs:enumeration value="All"/>
2749     </xs:restriction>
2750   </xs:simpleType>
2751 </xs:attribute>

```

```
2752 <xs:attribute name="objectType" type="xs:NCName"/>
2753 <xs:attribute name="predefined" type="xs:string"/>
2754 <xs:attributeGroup name="selectQualif">
2755   <xs:attribute ref="dst:objectType" use="optional"/>
2756   <xs:attribute ref="dst:predefined" use="optional"/>
2757 </xs:attributeGroup>
2758 <!--endsec(select)-->
2759 <!--sec(resquery)-->
2760 <xs:complexType name="ResultQueryBaseType">
2761   <xs:sequence>
2762     <xs:element ref="dst:ChangeFormat" minOccurs="0" maxOccurs="2"/>
2763   </xs:sequence>
2764   <xs:attributeGroup ref="dst:selectQualif"/>
2765   <xs:attribute ref="lu:itemIDRef" use="optional"/>
2766   <xs:attribute name="contingency" use="optional" type="xs:boolean"/>
2767   <xs:attribute name="includeCommonAttributes" use="optional" type="xs:boolean" default="0"/>
2768   <xs:attribute name="changedSince" use="optional" type="xs:dateTime"/>
2769   <xs:attribute ref="lu:itemID" use="optional"/>
2770 </xs:complexType>
2771 <xs:attributeGroup name="ItemDataAttributeGroup">
2772   <xs:attribute ref="lu:itemIDRef" use="optional"/>
2773   <xs:attribute name="notSorted" use="optional">
2774     <xs:simpleType>
2775       <xs:restriction base="xs:string">
2776         <xs:enumeration value="Now"/>
2777         <xs:enumeration value="Never"/>
2778       </xs:restriction>
2779     </xs:simpleType>
2780   </xs:attribute>
2781   <xs:attribute ref="dst:changeFormat" use="optional"/>
2782 </xs:attributeGroup>
2783 <!--endsec(resquery)-->
2784 <!--sec(testitem)-->
2785 <xs:complexType name="TestItemBaseType">
2786   <xs:attributeGroup ref="dst:selectQualif"/>
2787   <xs:attribute name="id" use="optional" type="xs:ID"/>
2788   <xs:attribute ref="lu:itemID" use="optional"/>
2789 </xs:complexType>
2790 <xs:element name="TestResult" type="dst:TestResultType"/>
2791 <xs:complexType name="TestResultType">
2792   <xs:simpleContent>
2793     <xs:extension base="xs:boolean">
2794       <xs:attribute ref="lu:itemIDRef" use="required"/>
2795     </xs:extension>
2796   </xs:simpleContent>
2797 </xs:complexType>
2798 <!--endsec(testitem)-->
2799 <!--sec(pagination)-->
2800 <xs:attributeGroup name="PaginationAttributeGroup">
2801   <xs:attribute name="count" use="optional" type="xs:nonNegativeInteger"/>
2802   <xs:attribute name="offset" use="optional" type="xs:nonNegativeInteger" default="0"/>
2803   <xs:attribute name="setID" use="optional" type="lu:IDType"/>
2804   <xs:attribute name="setReq" use="optional">
2805     <xs:simpleType>
2806       <xs:restriction base="xs:string">
2807         <xs:enumeration value="Static"/>
2808         <xs:enumeration value="DeleteSet"/>
2809       </xs:restriction>
2810     </xs:simpleType>
2811   </xs:attribute>
2812 </xs:attributeGroup>
2813 <xs:attributeGroup name="PaginationResponseAttributeGroup">
2814   <xs:attribute name="remaining" use="optional" type="xs:integer"/>
2815   <xs:attribute name="nextOffset" use="optional" type="xs:nonNegativeInteger" default="0"/>
2816   <xs:attribute name="setID" use="optional" type="lu:IDType"/>
2817 </xs:attributeGroup>
2818 <!--endsec(pagination)-->
```

```
2819 <!--sec(create)-->
2820 <xs:attributeGroup name="CreateItemAttributeGroup">
2821   <xs:attribute ref="dst:objectType" use="optional"/>
2822   <xs:attribute name="id" use="optional" type="xs:ID"/>
2823   <xs:attribute ref="lu:itemID" use="optional"/>
2824 </xs:attributeGroup>
2825 <!--endsec(create)-->
2826 <!--sec(mod)-->
2827 <xs:attributeGroup name="ModifyItemAttributeGroup">
2828   <xs:attributeGroup ref="dst:selectQualif"/>
2829   <xs:attribute name="notChangedSince" use="optional" type="xs:dateTime"/>
2830   <xs:attribute name="overrideAllowed" use="optional" type="xs:boolean" default="0"/>
2831   <xs:attribute name="id" use="optional" type="xs:ID"/>
2832   <xs:attribute ref="lu:itemID" use="optional"/>
2833 </xs:attributeGroup>
2834 <!--endsec(mod)-->
2835 <!--sec(del)-->
2836 <xs:complexType name="DeleteItemBaseType">
2837   <xs:attributeGroup ref="dst:selectQualif"/>
2838   <xs:attribute name="notChangedSince" use="optional" type="xs:dateTime"/>
2839   <xs:attribute name="id" use="optional" type="xs:ID"/>
2840   <xs:attribute ref="lu:itemID" use="optional"/>
2841 </xs:complexType>
2842 <xs:complexType name="DeleteResponseType">
2843   <xs:complexContent>
2844     <xs:extension base="lu:ResponseType"/>
2845   </xs:complexContent>
2846 </xs:complexType>
2847 <!--endsec(del)-->
2848 </xs:schema>
2849
2850
```

2851 **References**

2852 **Normative**

- 2853 [LibertyDisco] Hodges, Jeff, Cahill, Conor, eds. "Liberty ID-WSF Discovery Service Specification," Version 2.0,
2854 Liberty Alliance Project (30 July, 2006). <http://www.projectliberty.org/specs>
- 2855 [LibertySOAPBinding] Hodges, Jeff, Kemp, John, Aarts, Robert, Whitehead, Greg, Madsen, Paul, eds. "Lib-
2856 erty ID-WSF SOAP Binding Specification," Version 2.0, Liberty Alliance Project (30 July, 2006).
2857 <http://www.projectliberty.org/specs>
- 2858 [LibertyPAOS] Aarts, Robert, Kemp, John, eds. "Liberty Reverse HTTP Binding for SOAP Specification," Version
2859 2.0, Liberty Alliance Project (30 July, 2006). <http://www.projectliberty.org/specs>
- 2860 [LibertyInteract] Aarts, Robert, Madsen, Paul, eds. "Liberty ID-WSF Interaction Service Specification," Version 2.0,
2861 Liberty Alliance Project (30 July, 2006). <http://www.projectliberty.org/specs>
- 2862 [LibertySecMech] Hirsch, Frederick, eds. "Liberty ID-WSF Security Mechanisms Core," Version v2.0, Liberty
2863 Alliance Project (30 July, 2006). <http://www.projectliberty.org/specs>
- 2864 [LibertyGlossary] Hodges, Jeff, eds. "Liberty Technical Glossary," Version v2.0, Liberty Alliance Project (30 July,
2865 2006). <http://www.projectliberty.org/specs>
- 2866 [LibertyReg] Kemp, John, eds. "Liberty Enumeration Registry Governance," Version 1.1, Liberty Alliance Project (14
2867 December, 2004). <http://www.projectliberty.org/specs>
- 2868 [Schema1-2] Thompson, Henry S., Beech, David, Maloney, Murray, Mendelsohn, Noah, eds. (28 October
2869 2004). "XML Schema Part 1: Structures Second Edition," Recommendation, World Wide Web Consortium
2870 <http://www.w3.org/TR/xmlschema-1/>
- 2871 [RFC2119] S. Bradner "Key words for use in RFCs to Indicate Requirement Levels," RFC 2119, The Internet
2872 Engineering Task Force (March 1997). <http://www.ietf.org/rfc/rfc2119.txt>
- 2873 [XML] Bray, Tim, Paoli, Jean, Sperberg-McQueen, C. M., Maler, Eve, Yergeau, Francois, eds. (04 February 2004).
2874 "Extensible Markup Language (XML) 1.0 (Third Edition)," Recommendation, World Wide Web Consortium
2875 <http://www.w3.org/TR/2004/REC-xml-20040204>