International

Virtual

Observatory

Alliance

IVOA Data Access Layer Interface Version 1.0

IVOA Internal Working Draft 2010-11-24

Interest/Working Group:

http://www.ivoa.net/cgi-bin/twiki/bin/view/IVOA/IvoaDAL

This version:

WD-DALI-1.0-20101124

Latest version:

Not yet issued

Previous version(s):

Editors:

TBD

Authors:

TBD

Abstract

This document describes the Data Access Layer Interface (DALI). DALI defines the base web service interface common to all Data Access Layer (DAL) services. This standard defines the behaviour of common resources, the meaning and use of common parameters, success and error responses, and DAL service registration. The goal of this specification is to define the common elements that are shared across DAL services in order to encourage (require?) consistency across concrete DAL service specifications and to enable standard re-usable client and service implementations and libraries to be written and widely adopted.

Status of This Document

This is a working draft internal to the DAL-WG.

This is an IVOA Working Draft for review by IVOA members and other interested parties. It is a draft document and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use IVOA Working Drafts as reference materials or to cite them as other than "work in progress".

A list of current IVOA Recommendations and other technical documents *can be found at http://www.ivoa.net/Documents/.*

Acknowledgements

"Ack here, if any"

Contents

1 Introduction

The Data Access Layer Interface (DALI) defines resources, parameters, and responses common to all DAL services so that concrete DAL service specifications need not repeat these common elements.

1.1 General Principles

TODO: define scope and context, mainly by referring background material like DAL2 note, etc.

1.2 The Role in the IVOA Architecture

TODO: include architecture diagram and show where DALI fits in there

TODO: elaborate on related standards, e.g.: UWS, VOResource, VOSI, VOTable,

1.3 Document Roadmap

TODO: brief summary of subsequent sections once they are nailed down

2 Resources

DAL services are implemented as HTTP REST [ref] web services. The primary resource in a DAL service is a job. A DAL job is defined by parameters (see 3) and can be executed either synchronously or asynchronously. A concrete service specification defines the job parameters and the manner of execution is defined by separate resources below.

In addition to job list resources, DAL services also implement several Virtual Observatory Support Interface (VOSI) resources to describe service availability, capabilities, and content.

A concrete DAL service must define at least one DALI-async or DALI-sync resource. It may define both with the same job semantics (e.g. TAP-1.0) or it may define one with one kind of job and the other with a separate kind of job (a service that does some things synchronously and others asynchronously).

resource type	resource name	required
DALI-async	service specific	service specific
DALI-sync	service-specific	service specific
VOSI-availability	/availability	yes
VOSI-capabilities	/capabilities	yes
VOSI-tables	/tables	service specific

A simple query-only DAL service like ConeSearch can be easily described as having a single DALI-sync resource where the job is a query and the response is the result of the query. Slightly more complex services like SIA and SSA ...

2.1 Asynchronous Execution: DALI-async

Asynchronous resources are resources that represent a list of asynchronous jobs as defined by the Universal Worker Service (UWS) pattern [ref]. Requests can create, modify, and delete jobs in the job list. Special requests to modify the phase of the job cause the job to execute or abort.

As specified in UWS, a job is created by using the HTTP POST method to modify the job list. The response will always be an HTTP redirect (status code 303) and the Location (HTTP header) will contain the URL to the job (a child resource of the job list).

```
POST http://example.com/base/async-jobs
< HTTP/1.1 303 See Other
< Location: http://example.com/base/async-jobs/123
```

The job description (an XML document defined by the UWS schema) can always be retrieved by accessing the job URL with the HTTP GET method:

```
GET http://example.com/base/async-jobs
[sample UWS xml job description here]
```

In addition to the UWS job metadata, DAL jobs are defined by a set of parameter-value pairs. The client may add new parameters by modifying the current list of parameters via the HTTP POST method:

```
POST FOO=bar http://example.com/base/async-jobs/123/parameters
```

The UWS standard allows parameters to be POSTed along with the initial job-creation request, or POSTed to the job URL, or POSTed to the parameter list (the parameters child resource) directly (as in the above example). This is easily implemented by simply applying all UWS-specific parameters to the UWS job itself and putting all remaining parameters into the parameter list.

```
TODO: examples of all resources one can post DAL parameters to
```

A concrete DAL service specification will specify one or more asynchronous job-list resources and whether they are mandatory or optional. It may mandate a specific resource name to support simple client use, or it can allow the resource name to be described in the service metadata (see 2.4).

2.2 Synchronous Execution: DALI-sync

Synchronous resources are resources that accept a request (a DAL job description) and return the response (the result) directly. Synchronous requests can be made using either the HTTP GET or POST method. The parameters used to specify the job are the same for synchronous and asynchronous DAL jobs. A synchronous job is created by a GET or POST request to a synchronous job list, executed automatically, and the result returned in the response. The web service is permitted to split the operation of a synchronous request into multiple HTTP requests as long as it is transparent to standard clients. This means that the service may use HTTP redirects (status code 302 or 303) and the Location header to execute a synchronous job in multiple steps. For example, a service may

- immediately execute and return the result in the response, or
- the response is an HTTP redirect (status code 303) and the Location (HTTP header) will contain a URL; the client accesses this URL with the HTTP GET method to execute the job and get the result

Clients should generally expect to get redirects and follow them in order to complete requests.

A concrete DAL service specification will specify one or more synchronous job-list resources and whether they are mandatory or optional. It may mandate a specific resource name to support simple client use, or it can allow the resource name to be described in the service metadata (see 2.4).

2.3 Availability: VOSI-availability

VOSI-availability [ref] defines a simple web resource that reports on the current ability of the service to perform. In DAL services, this resource is always accessed as a resource named *availability* that is a child of the base URL for the service.

All DAL services must implement the /*availability* resource.

TODO: put example XML here

2.4 Capabilities: VOSI-capabilities

VOSI-capabilities [ref] defines a simple web resource that returns an XML document describing the service. In DAL services, this resource is always accessed as a resource named *capabilities* that is a child of the base URL for the service. The VOSI-capabilities described all the resources exposed by the service, including which standards each resource implements.

All DAL services must implement the /*capabilities* resource.

TODO: put basic example output here? probably [xref] to section on service registration

2.5 Content: VOSI-tables

VOSI-tables [ref] defines a simple web resource that returns an XML document describing the content of the service. In DAL services which include it, this resource is always accessed as a resource named *tables* that is a child of the base URL for the service. The document format is defined by the VODataService [ref] standard and allows the service to describe their content as a tableset: schemas, tables, and columns. TODO: example tables document

A concrete DAL service specification will specify if the /*tables* resource is mandatory or optional.

TODO: put simple example XML here?

3 Parameters

A DAL job is defined by a set of parameter-value pairs. Some of these parameters are standard meaning and are defined here, but most are defined by the service specification or another standard (e.g. PQL [ref]).

3.1 REQUEST

The REQUEST parameter specifies the type of the DAL job at the highest level. In many cases, a service will have only one possible value (e.g. TAP-1.0 only supports REQUEST=doQuery), but this parameter is still used in such cases as future versions or non-standard (site-specific features) may support additional values.

3.2 VERSION

The VERSION parameter is used so the client can specify which version of the service standard they are using to make the request. This allows implementers to support multiple versions of a standard in a single web service and with a single resource for the DAL job list.

TODO: put version negotiation rules here

3.3 FORMAT

The FORMAT parameter is used so the client can specify the format of the response (e.g. the output of the job). While the list of supported values are specific to a concrete service specification, the general usage is to support values that are content-types (mimetypes) for known formats as well as shortcut symbolic values.

table type	MIME type(s)	short form
VOTable	application/x-votable+xml text/xml	votable
comma separated values	text/csv	csv
tab separated values	text/tab-separated-values	tsv
FITS file	application/fits	fits
pretty-printed text	text/plain	text
pretty-printed Web page	text/html	html

A DAL service **must** accept a *FORMAT* parameter indicating a format that the service supports and **should** fail (xref to error handling section) where the

FORMAT parameter specifies a format not supported by the service implementation.

A concrete DAL service specification will specify any mandatory or optional formats as well as new formats not listed above; it may also place limitations on the structure for formats that are flexible. For example, a resource that responds with tabular output only may impose a limitation that FITS files only contain FITS tables, possibly only of specific types (ascii or binary).

If a client requests a format by specifying the mimetype (as opposed to one of the short forms), the response that delivers that content must set that mimetype in the Content-Type header. This is only an issue when a format has multiple acceptable mimetypes (e.g. VOTable).

DAL services are free to support custom formats by accepting non-standard values for the FORMAT parameter.

3.4 MAXREC

For resources performing discovery (querying for an arbitrary number of records), the resource **must** accept a *MAXREC* parameter specifying the maximum number of records to be returned. If *MAXREC* is not specified in a request, the service **may** apply a default value or **may** set no limit. If the size of the result exceeds this value, the service **must** only return the requested number of rows. If the result set is truncated in this fashion, it must include an overflow indicator as specified in 4.3 .

The service **must** support the special value of *MAXREC=0.* This value indicates that, in the event of an otherwise valid request, a valid response be returned containing metadata, no results, and an overflow indicator as above. The service is not required to execute the request and the overflow indicator does not necessarily mean that there is at least one record satisfying the query. The service **may** perform validation and may try to execute the request, in which case a MAXREC=0 request can fail.

3.5 RUNID

The service **should** implement the *RUNID* parameter, used to tag service requests with identifier of a larger job of which the request may be part. The *RUNID* parameter is defined in [ref].

For example, if a cross match portal issues multiple requests to remote services to carry out a cross-match operation, all would receive the same *RUNID*, and the service logs could later be analyzed to reconstruct the service operations initiated in response to the job.

The service **should** ensure that *RUNID* is preserved in any service logs and **should** pass on the *RUNID* value in any calls to other services.

3.6 Case of Parameters

Parameter names are not case sensitive; a DAL service must treat upper-, lower-, and mixed-case parameter names as equal. Parameter values are case sensitive unless a concrete DAL service specification explicitly states that the values of a specific parameter are to be treated as case-insensitive. For example, the following are equivalent: *FOO=bar*, *Foo=bar*, *foo=bar*. Unless explicitly stated by the service specification, these are not equivalent: FOO=*bar*, FOO=*Bar*, FOO=*BAR*.

In this document, parameter names are typically shown in uppercase for typographical clarity, not as a requirement.

3.7 Order and Cardinality of Parameters

Parameters in a request **may** be specified in any order.

When request parameters are duplicated with conflicting values, the response from the service is undefined. The service **may** reject the request or it **may** pick one value for the parameter. Clients **should not** repeat parameters in a request. If a parameter has multiple values, a single parameter-value pair must be used and the value must use the list syntax in 3.12 .

> *TBD: Forms in web apps that allow selection of multiple values do include the same parameter name multiple times, once for each value posted. We should discuss this prohibition as it makes it hard to use simple web forms directly.*

3.8 Parameter Indirection

The value of any parameter can be taken indirectly from an external object rather than specified directly, if specified in the description of an individual parameter. The '@' character is used to denote this symbolic reference:

```
FOO=@something
```

The meaning or interpretation of the referenced symbolic value ('@' target) is defined by the individual parameter.

> *TBD: Typical use is to refer to a list or table of values that is also included with the request, e.g. referred to in a separate parameter or included as inline content. May need some guidance/examples to clarify what this does.*

3.9 Missing or null-valued parameters

If a parameter is not included in a query its value is unset; no value has been specified. If a parameter is given a null value, e.g., "...&FOO=&...", the parameter value has been set and the value is the null string. Whether or not a null parameter value is significant is defined by the individual parameter. If only the parameter name is given, e.g., "...&FOO&...", it is the same as if the parameter was not specified, and the parameter value is unset.

TBD: Are there any practical cases where we need to differentiate between not set and set to null?

3.10 Literal Values: Numbers, Boolean, Date, and Time

Integer numbers **must** be represented in a manner consistent with the specification for integers in *XML Schema Datatypes* [10].

Real numbers **must** be represented in a manner consistent with the specification for double-precision numbers in *XML Schema Datatypes*. This representation allows for integer, decimal and exponential notations.

Boolean values must be represented in a manner consistent with the specification for Boolean in XML Schema Datatypes. The values *0* and *false* are equivalent. The values *1* and *true* are equivalent.

```
FOO=1
FOO=true
```

```
BAR=0
BAR=false
```

Absence of an optional value is equivalent to false.

Date and time values must be represented as ISO 8601 formatted strings with a T character separating the date and time components. Fractions of a second are permitted but not required. For example:

```
2000-01-02T15:20:30
```

is January 2, 2000 at 3:20 PM (plus 30 seconds). Services must interpret all date and time values as UTC [ref].

3.11 Range Values

Parameters thats specify a range of values use the forward slash ("/") character as the separator between elements of the range specification (as in the ISO 8601 date specification after which this convention is patterned). For example,a range consisting of all values from 5E-7 to 8E-7 inclusive would be:

```
FOO=5E-7/8E-7
```

If a third field is specified it is a step size for traversing the indicated range, such as the above range in steps of 1E-8 (30 steps):

```
FOO=5E-7/8E-7/1E-8
```

If a parameter permits a step size the semantics of the step size are defined by the specific parameter.

An open range may be specified by omitting either range value. If the first value is omitted the range is open toward lower values. If the second value is omitted the range is open toward higher values. Omitting both values indicates an infinite range which accepts all values. For example, an open range which accepts all values less than or equal to 5 would be encoded as shown below:

```
FOO=/5
```

Range values can only be used with parameters which specify numeric and date values.

TBD: String ranges could potentially be defined, but advanced string processing would probably require a different and more complex facility. Also, when used with strings one would need to specify a way to escape the separator character in values, which gets ugly.

3.12 List Values

Parameters which are multi-valued (a list of values) use the comma (“,”) as the separator between successive items in the list. Embedded white space is not permitted. The values in the list may be of any data type (3.10), including ranges (3.11). For example, the list of values including A and B would be:

```
FOO=A,B
```

The order of values in a list may be arbitrary or important depending on the parameter. For example, one parameter may be defined as matching any value (logical OR), e.g.:

```
GENRE=rock,classical,folk
```

in which case the order does not matter, while another parameter may be defined with some structure, e.g.:

```
POS=20,30
```

where the two values together are coordinates (longitude and latitude) and the order does matter. The importance and meaning of the order of values in a list is specified for each parameter.

The range and list syntax may be used together to specify a list of ranges, e.g.:

```
FOO=1/2,5/6
```

In some lists, individual entries may be empty, and **should** be represented by the empty string. Thus, two successive commas indicate an empty item, as does a leading comma or a trailing comma. An empty list **should** be interpreted either as a list containing no items, or as a list containing a single empty item, depending upon the context.

3.13 Qualifiers

If specified by the definition of a particular parameter, a single-valued parameter, range, or list **may** be qualified by appending a semicolon (";") followed by a qualifier string. This could be used to specify an alternate coordinate system, e.g.:

```
POS=180.0,1.0;GALACTIC
```

could specify a position in galactic coordinates. In some cases, multiple semicolons may be used to delimit separate sub-lists or clauses within the parameter value.

4 Responses

All DAL service requests eventually result in one of three kinds of responses: successful HTTP status code (200) and a service- and resource-specific representation of the results, an HTTP status code (??) and an unspecified error document, or a redirect HTTP status code (302 or 303) with a URL in the HTTP header.

4.1 Successful Requests

Successfully executed requests should result in a response with HTTP status code 200 (OK) and a response in the format requested by the client (see 3.3) or in the default format for the service. The service should set the following HTTP headers to the correct values where possible.

Content-Type	mimetype of the response
Content-Encoding	encoding/compression of the response
Content-Length	size of the response in bytes (generally not known for dynamically generated and streamed response)
Last-Modified	timestamp when the resource was last changed (not applicable to dynamically generated response)

For jobs executed using a DALI-async resource, the result(s) **must** be made available as child resources of the result list and directly accessible there. For jobs that inherently create a single result, this result **must** be named *result* in the result list and be directly accessible by that name, e.g.:

```
GET http://example.com/base/joblist/123/results/result
```

For concrete DAL service specifications where multiple result files may be produced, the specification may dictate the names or it may leave it up to implementations to chose suitable names.

TBD: In TAP we specfied that the query result was named result explicitly, but then when people/apps download the file it may end up with just that name in the local filesystem and thus collide with other downloaded TAP results. This can be avoided on the server-side by using a redirect to a URL that ends in a sensible filename and/or setting the Content-Disposition header... would it be better to always let the implementor name the result with a suitable (file) name directly?

4.2 Errors

If the service detects an exceptional condition, it **must** return an error document with an appropriate HTTP-status code. DAL services distinguish three classes of errors:

- Errors in the use of the HTTP protocol
- Errors in the use of the specific DAL protocol, including an invalid request
- Errors caused by a failure of the service to complete a valid request

Error documents for HTTP-level errors are not specified since responses to these errors may be generated by service containers and cannot be controlled by service implementations. There are several cases where a DAL service could return an HTTP error. First, a DALI-async resource could return a 404 (not found) error if the client accesses a job within the UWS joblist that does not exist, or accesses a child resource of the job that does not exist (e.g. the error resource of a job that has not run and failed, or a specific result resource in the result list that does not exist). Second, access to a resource could result in an HTTP 401 (not authorized) response if authentication is required or an HTTP 403 (forbidden) error if the client is not allowed to access the requested resource.

Error documents describing errors in use of the DAL service protocol **must** be VOTable documents as described in [xref]; any result-format specified in the request is ignored. In all cases, these are errors that occur when the job is executed and do not override any error behaviour for a UWS resource which specifies the behaviour and errors associated with interacting with the job itself.

If the invalid job is being executed using a DALI-async resource, the error document must be accessible from the <DALI-async>/<jobid>/error resource (specified by UWS) and when accessed via that resource it must be returned with an HTTP status code should be 200, e.g.:

```
GET http://example.com/base/joblist/123/error
```

If the error document is being returned directly after a DALI-sync request, the service must use HTTP status code 200 (successfully returning a response to the request).

Error documents describing the failure of the service to execute a valid job are returned as above, but the VOTable document must ??? to indicate that the request was valid but failed due to (i) an internal limitation or (ii) a transient failure and may succeed in the future.

TBD: How do we generically divide errors into these 3 types: invalid, internal limitation/refusal, or transient failure?

4.3 Overflows

If a request is executed by a DAL service, the number of records in the results may exceed a limit requested by the user (using the MAXREC parameter) or a limit set by the service implementation (the default or maximum value of MAXREC). In these cases, the request is said to have 'overflowed'. Typically, a service will not detect an overflow until some part of the results have already been sent to the client.

If an overflow occurs, the service **must** produce a table of results that is valid, in the required output format, and which contains all the results up to the point of overflow. Since an output overflow is not an error condition, the MIME type and the HTTP status-code of the response **must** be the same as for any successful request.

If the output format is VOTable, section Error: Reference source not found describes the method by which the overflow is reported. No method of reporting an overflow is defined for formats other than VOTable.

4.4 Redirection

A concrete DAL service specification may require that HTTP redirects (302 or 303) be used to communicate the location of an alternate resource which should be accessed by the client via the HTTP GET method. For example, the UWS pattern used for DALI-async (2.1) requires this behaviour. Even when not required, concrete DAL service specifications must allow implementors to use redirects and clients should follow these redirects using normal HTTP semantics [ref].

4.5 Use of VOTable

VOTable is a general format. In DAL services we require that it be used in a particular way. The result VOTable document **must** comply with VOTable v1.2 or greater [ref]. For columns containing coordinate values, the coordinate system metadata should be provided as described in [ref].

For resources where the job is a query, the VOTable **must** contain a *RESOURCE* element identified with the attribute *type="results"*, containing a single *TABLE* element with the results of the query. Additional *RESOURCE* elements may be present, but the usage of any such elements is not defined here.

The *RESOURCE* element **must** contain, before the *TABLE* element, an *INFO* element with attribute *name = "QUERY_STATUS"*. The *value* attribute **must** contain one of the following values:

1. “OK”, meaning that the query executed successfully and a result table is included in the resource

2. “ERROR”, meaning that an error was detected at the level of the TAP protocol or the query failed to execute

The content of the INFO element conveying the status **should** be a message suitable for display to the user describing the status.

```
<INFO name="QUERY_STATUS" value="OK"/>
```

```
<INFO name="QUERY_STATUS" value="OK">Successful query</INFO>
```

```
<INFO name=”QUERYY_STATUS" value="ERROR">
value out of range in POS=45,91
</INFO>
```

Additional *INFO* elements **may** be provided, e.g., to echo the input parameters back to the client in the query response (a useful feature for debugging or to self-document the query response), but clients **should not** depend on these.

```
<RESOURCE type=”results”>
<INFO name=”QUERY_STATUS” value=”ERROR”>
missing REQUEST parameter
</INFO>
<INFO name=”SPECIFICATION” value=”TAP”/>
<INFO name=”VERSION” value=”1.0”/>
...
</RESOURCE>
```

If an overflow occurs (result exceeds MAXREC), the service must close the table and append another INFO element to the RESOURCE (after the TABLE) with *name=”QUERY_STATUS”* and the value=”*OVERFLOW”.*

```
<RESOURCE type=”results”>
<INFO name=”QUERY_STATUS” value=”OK”/>
...
<TABLE>...</TABLE>
<INFO name=”QUERY_STATUS” value=”OVERFLOW”/>
</RESOURCE>
```

In the above example, the TABLE should have exactly MAXREC rows.

If an error occurs while writing the rows of the VOTable, the service must close the table and append another INFO element to the RESOURCE, after the

TABLE, with *name=”QUERY_STATUS”* and the value=”ERROR*”.*

```
<RESOURCE type=”results”>
<INFO name=”QUERY_STATUS” value=”OK”/>
...
<TABLE>...</TABLE>
<INFO name=”QUERY_STATUS” value=”ERROR”>
unexpected IO error while converting something
</INFO>
</RESOURCE>
```

The content of these trailing INFO elements is optional and intended for users; client software **should not** depend on it.

Thus, one INFO element with *name=”QUERY_STATUS”* and *value=”OK”* or *value=”ERROR”* **must** be included before the TABLE. If the TABLE does not contain the entire result, one INFO element with *value=”OVERFLOW”* or *value=”ERROR”* **must** be included after the table.

TBD: QUERY_STATUS seems meaningful when the job is a query (e.g. for data discovery or querying TAP or ConeSearch), but it is not completely generic... or it is implying the HTTP query string of a simple HTTP GET request, which is also no longer the only scenario. Is it worth defining a more general name for status at the expense of making SIAv2 and SSAv2 different from v1?

5 References

[1] D. Tody, F. Bonnarel, M. Dolensky, J. Salgado, DAL-WG, *IVOA Data Access Layer Service Architecture and Standard Profile,* IVOA Note 5 October 2008. http://www.ivoa.net/internal/IVOA/SiaInterface/DAL2_Architecture.pdf

[2] S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels,* IETF RFC 2119. http://www.ietf.org/rfc/rfc2119.txt

[3] T. Berner-Lee, R. Fielding L. Masinter, *Uniform Resource Identifiers (URI): Generic Syntax,* IETF RFC 2396. http://www.ietf.org/rfc/rfc2396.txt

[4] P. Biron & A. Malhotra, *XML Schema Part 2: Datatypes Second Edition,* W3C Recommendation 28 October 2004. http://www.w3.org/TR/xmlschema-2/

[5] R. Fielding, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, *Hypertext Transfer Protocol – HTTP/1.1*, IETF RFC 2616. http://www.rfc-editor.org/rfc/rfc2616.txt

[6] N. Freed & N. Borenstein, *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies,* IETF RFC 2045.

http://www.ietf.org/rfc/rfc2045.txt

[7] Y. Shafranovich, *Common Format and MIME Type for Comma-Separated Values (CSV) Files*, IETF RFC 4180.

http://www.ietf.org/rfc/rfc4180.txt

[8] IANA, *MIME Media Types*,

http://www.iana.org/assignments/media-types/text/tab-separated-values

[9] GWS-WG, G. Rixon (ed.), *IVOA Support Interfaces Version 1.00*, IVOA Working Draft, 25 August 2009. http://www.ivoa.net/Documents/VOSI/1.0

[10] P. Harrison & G. Rixon, *Universal Worker Service Version 1.0,* IVOA Proposed Recommendation, 09 September 2009. http://www.ivoa.net/Documents/UWS/1.0

[11] F. Ochsenbein (ed.), R. Williams, *VOTable Format DefinitionVersion 1.2*, IVOA Proposed Recommendation 29 September 2009. http://www.ivoa.net/Documents/VOTable/1.2

[12] R, Plante (ed.), A. Stébé, K. Benson, P. Dowler, M. Graham, G. Greene, P. Harrison, G. Lemson, A. Linde, G. Rixon & IVOA Registry-WG, *VODataService: a VOResource Schema Extension for Describing Collections and Services Version 1.1.* IVOA Proposed Recommendation, 03 September 2009
http://www.ivoa.net/Documents/VODataService/1.1

[13] P. Dowler, G. Rixon, D. Tody, DAL-WG, *Table Access Protocol,* IVOA Recommendation 27 March 2010. http://www.ivoa.net/Documents/TAP/1.0