



*International*

*Virtual*

*Observatory*

*Alliance*

## **UWS recast as a REST protocol**

**Version 1.00**

***IVOA Note 2007 February 26***

**This version:**

1.00-20070226

**Latest version:**

None.

**Previous version(s):**

None.

**Author(s):**

Guy Rixon

---

### **Abstract**

Universal Worker Service is a pattern for asynchronous web-services. Early drafts of the pattern were based on SOAP. This note presents the features of UWS in a REST protocol.

## Status of This Document

*This is an IVOA Note expressing suggestions from and opinions of the authors. It is intended to share best practices, possible approaches, or other perspectives on interoperability with the Virtual Observatory. It should not be referenced or otherwise interpreted as a standard specification.*

A list of [current IVOA Recommendations and other technical documents](http://www.ivoa.net/Documents/) can be found at <http://www.ivoa.net/Documents/>.

## Acknowledgements

I thank Norman Gray for pointing out that UWS could be done as a REST protocol and for being patient when I was slow to see the benefits.

## Contents

- [1 Introduction](#)
- [2 The absolute minimum](#)
- [3 Extra features](#)
  - [3.1 Canceling a job](#)
  - [3.2 Negotiating the job lifetime](#)
  - [3.3 Getting multiple, named results](#)
  - [3.4 Job estimation and two-stage job-launching](#)
- [4 Reprise](#)
- [5 Web browser interface](#)
- [6 Applications](#)
  - [6.1 Asynchronous SIAP](#)
  - [6.2 UWS-PA](#)
  - [6.3 Asynchronous ADQL-query](#)
- [7 Conclusion](#)
- [8 References](#)

## 1 Introduction

The Universal Worker Service (UWS) pattern defines a pattern for services that work asynchronously with respect to their client. These service can support long-running operations.

The existing drafts of the UWS standard [1] presume it to be a SOAP protocol. This follows from its origins: it is based on the Common Execution Connector interface (CEC) in AstroGrid's Common Execution Architecture (CEA) which uses SOAP, and was conceived to use the features of Web Services Resource Framework (WS-RF) which is an extension to SOAP.

It nows seems that many groups are inventing asynchronous protocols based on HTTP; discussions between AstroGrid, Caltech and SDSC identified three such

cases. These are local protocols for use inside an installation rather than registered, IVOA services. They are simple protocols and do not involve SOAP.

Can UWS be recast as a REST protocol? Certainly; any SOAP service can be redesigned as a REST service. Would a RESTful version of UWS be better for the IVOA than a SOAP version? Would a RESTful UWS be simple enough to use for the little, local web-services? We cannot tell until we see the protocol details. This note describes a possible REST protocol for UWS.

## 2 The absolute minimum

Any asynchronous protocol needs three things.

1. A way to start a job.
2. A way to find out whether the job has finished (or failed).
3. A way to get the results of the job.

Starting a job changes the state of the service, so needs to be done with an HTTP post request rather than an HTTP get; c.f. the simple, synchronous protocols of IVOA such as SSAP. Therefore, there must be a resource, with a URI, in the service to which job requests can be posted: call it */jobs*. The content of the posted request depends on the kind of service and the jobs it accepts.

To find out the state of the job, or to get the results, the client needs to identify the job. Let's presume that it has some simple job-number that is unique within the service and use 42 as an example. Therefore the result of a successful post to */jobs* is the creation of a new resource, */jobs/42*. The client can work with this to get information about the job. We presume that the response message when a job is posted states the number of the created job.

There are two things to get back from the job resource: whether or not it is finished and the results of its processing. REST principles state that these things should be represented as resources with their own URIs. We need to associate them with their parent job, so we call them */jobs/42/phase* and */jobs/42/results*. A get request to either of these retrieves the information.

This is sufficient for a minimal protocol:

1. post to */jobs* to start the job; read the job number from the response.
2. get */jobs/42/phase* repeatedly until the state changes from "working" to "finished" or "failed";
3. get */jobs/042/results*.

All asynchronous HTTP protocols end up like this; only the detailed layout of the resources changes. This particular pattern can be used in simple, local protocols and can be the basis of IVOA protocols such as UWS.

### 3 Extra features

UWS v0.2 [1] specifies several features in addition to the minimal protocol described above.

#### 3.1 Canceling a job

The job is represented as a resource, */jobs/042* in the example above. Therefore, the natural way to cancel it is to delete  $\square$  in the HTTP sense  $\square$  that resource. However, HTTP-delete is not universally supported, so it is convenient to allow the deletion also to be triggered by a posted request to the job resource.

#### 3.2 Negotiating the job lifetime

A job in UWS has a lifetime, initially chosen by the service but negotiable by the client. After the lifetime expires, the records of the job, including its results, are deleted from the service.

In the REST form, the lifetime can be a sub-resource of part the *state* resource. It's better not to include the lifetime inside the state resource as we'd like the state resource itself to be common to UWS and the minimal protocols. Suppose that *state/lifetime* contains an XML document of this form:

```
<termination-time>2007-02-22T00:00:00.0</termination-time>
```

Note that an absolute time is used rather than a relative lifetime. This eliminates the need to rewrite the document each time the client gets it ,and encourages HTTP caching.

To request a different termination time, the service could either require the client to put (i.e. HTTP put) a revised version of the XML document, or it could support a posted request with a termination-time parameter. Putting a state document is closer to the spirit of REST, but posting a parameterized change is probably easier for the client. A UWS should support both.

#### 3.3 Getting multiple, named results

Our minimal protocol treats all its results as one resource; by implication all the results have the same MIME type and can be streamed as a unit. Sometimes this is enough but a general protocol such as UWS has to support multiple results of arbitrary types.

The most RESTful way to do this is to represent each result as a separate resource, subordinate to the job resource itself. The names of these resources depend on the exact application. For example, consider an application that produces an image mosaic: its outputs are the main image, an image expressing coverage in the mosaic and a status log from the mosaicking code. Thus the client sees

<i>/jobs/42/results</i>	(application/xml)
<i>/jobs/42/mosaic</i>	(image/fits)
<i>/jobs/42/area-map</i>	(image/fits)
<i>/jobs/42/log</i>	(text/plain)

All these can be got, under their natural MIME-types, as separate requests. The results resource is an XML document giving the URIs for the named outputs.

### 3.4 Job estimation and two-stage job-launching

UWS v0.2 specifies that a job should be requested of a service in one operation and set running in a second. This two-stage process allows the client to ask the service for an estimate of the cost or processing time of the job before committing it.

In the REST form, the cost-and-time estimate can be handled as another resource under the job resource, e.g. */jobs/042/quote* in our persistent example. This resource is an XML document that can be got (but not posted put or deleted). Not all services can provide quotes; in these cases, a request to get the quote returns a 404 error.

At this level of analysis, the content of the quote document doesn't matter.

How does the client accept the quote and commit the job to execution? One way is to put an updated version of the quote document with some extra element that translated to "I accept". Another is to post a request to the quote URI. A UWS should support both.

## 4 Reprise

Our RESTful protocol now contains the following parts.

Resource	Result of HTTP-get	Result of HTTP-post	Result of HTTP-put	Result of HTTP-delete
<i>/jobs</i>	Returns list of jobs	Starts new job	N/A	N/A
<i>/jobs/(job-id)</i>	Returns links to sub-resources	Cancel job	N/A	Cancel job
<i>/jobs/(job-id)/phase</i>	Phase of job	N/A	N/A	N/A
<i>/jobs/(job-id)/results</i>	All job results in one document	N/A	N/A	N/A
<i>/jobs/(job-id)/(result-id)</i>	Single, named job-result	N/A	N/A	N/A
<i>/jobs/(job-id)/lifetime</i>	Termination time	Adjusts termination time	Adjusts termination time	N/A
<i>/jobs/(job-id)/quote</i>	Estimate of job cost and duration	Commits job	Commits job	N/A

The first four resources in the table are needed by any asynchronous REST protocol, although the post and delete operations on */jobs/(job-id)* may be left out

of the simplest protocols. The presence of named result-resources depends on the application. The lifetime and quote resources are specific to UWS.

The full sequence for a UWS job would be this:

1. post to `/jobs` to create a new job; get the job-number from the response;
2. get `/jobs/42/quote` to check that the proposed execution is reasonable;
3. post to `/jobs/42` to commit the job to execution;
4. get `/jobs/42/phase` until the job is finished;
5. get `/jobs/42/results` or other output.

The foregoing is not a *specification* of RESTful UWS. It is merely a demonstration that a RESTful UWS can be designed and might be useful.

## 5 Web browser interface

The RESTful protocol described above is intended to be driven by rich clients that specifically understand UWS. However, REST is the natural architecture of the web and web browsers support some parts of REST. Therefore, it is possible to build in an implicit UI for browser users.

To feed renderable HTML to browsers while presenting machine-readable XML to rich clients, the XML documents returns from a UWS should have processing instructions that link to XSLT stylesheets;<sup>1</sup> e.g.

```
<?xml-stylesheet type='text/xsl' href='../Job.xsl'?>
```

The limitations of REST for browsers are as follows.

- Web browsers don't support HTTP put or delete (because HTML forms do not support these operations).
- Web browsers behave badly if a user tries to reload the result of a posted request.

We avoid the first limitation by repeating the put and delete features as posted requests to particular resources. This adds a little to the cost of the service but the gain in usability is great.

We avoid the second limitation by redirecting a posted request, *after* executing the request, to another resource that can be got. E.g., posting a request for a new job creates the job and redirects to the resource for that job as a get request. A web browser will naturally follow the redirection. A rich client may follow the redirection but need not; it can, instead, simply read the *Location* header in the redirection and remember the URI for the job.

---

<sup>1</sup> Another way to do this might be to use HTTP content-negotiation such that browsers get HTML and rich clients get XML. It's simpler to send the XML with the stylesheet reference to both. Most modern browsers can handle XSLT stylesheets.

## 6 Applications

To illustrate and validate this design, I present three applications of RESTful UWS. In each case, applying UWS means specifying what is posted to create a job and how the outputs are arranged.

### 6.1 Asynchronous SIAP

An asynchronous version of Simple Image Access Protocol would be appropriate for the case where images have to be constructed by mosaicking or a similar, lengthy operation<sup>2</sup>. In this protocol, the mosaicking is done during the query operation, which is asynchronous to the client, and the mosaics are ready for download when the query completes.<sup>3</sup>

In this protocol, the posted request to create a job takes the parameters of a regular, synchronous, SIAP request.<sup>4</sup> The only output is the VOTable of images, and this is what is got from the *results* resource; there are no sub-resources of *results*.

### 6.2 UWS-PA

UWS-PA presumes that applications can be parameterized such that one kind of UWS service can provide access to many such applications. It is a standard protocol for “everything else” after special cases such as access to images and spectra have been given their own protocols. The applications are registered separately in the IVO registry and UWS-PA services register their ability to run particular applications. This approach follows from the AstroGrid/EuroVO Common Execution Architecture.[4]

To start a job, the client posts an XML document defining which application is to be run and stating its input and output parameters. These inputs and outputs can either be embedded in the document or referred to by URIs.

The available set of result resources depends on the application being run and varies from job to job. Getting the master *results* resource for a job returns an XML document listing the possible results. The individual result resources under *results* have the names given in the registration of the application.

### 6.3 Asynchronous ADQL-query

A query in Astronomical Data Query Language (ADQL) can take an arbitrary amount of time to execute. It is useful if there is an asynchronous service for queries.

---

2 E.g. Reconstructing radio images from visibility data-sets.

3 There is another approach for this use case: make the query on the image catalogue synchronous and only construct the mosaic when the client tries to download it. This latter approach has the advantage that mosaics are only computed when the user really wants to see them, but it has major disadvantages in control of the long-running mosaicking process.

4 They are sent as a URL-encoded string in the body of the post request. This means that an HTML form posting to the right resource can send the request.

To start an asynchronous query, the client posts a request to */jobs* in which the ADQL is a parameter.<sup>5</sup> This presumes that the request URI and the *SELECT* clause of the ADQL together identify the database to be queried.

The results are one file containing the output table. There are no sub-resources in the results.

## 7 Conclusion

Can UWS and local, simple, asynchronous protocols share a common core? Yes; I think the methods described above are naturally common to both and not onerous to use in a minimal protocol. A UWS client can use the resources provided in the minimal protocol and simply doesn't find the resources for the advanced parts of UWS.

Is RESTful UWS simpler than SOAP (as in UWS v0.2) for authors of clients? Probably; the REST client needs very little code. A basic client, say for testing a service, could be made just from HTML pages. A "proper", rich client, for embedding in an application, needs only basic support for HTTP and XML. The number of messages per job is the same for REST and for SOAP and the SOAP messages are slightly harder to form and parse. For developers who are not already equipped with a SOAP toolkit, REST is much easier.

Is RESTful UWS simpler than SOAP for authors of services? Probably not; SOAP toolkits make it easy to write services. Again, developers without toolkits will find REST much easier. Anybody who wants to write a minimal HTTP service and later add UWS support can do so fairly easily with REST; with SOAP, the service interface has to be rewritten entirely.

## 8 References

[1] *Asynchronous-activity proposals*, IVOA Grid and Web Services Working group, [http://www.ivoa.net/twiki/bin/view/IVOA/IvoaGridAndWebServices#Asynchronous\\_activities\\_proposal](http://www.ivoa.net/twiki/bin/view/IVOA/IvoaGridAndWebServices#Asynchronous_activities_proposal)

[2] *Why REST failed*, Elliotte R. Harold, <http://cafe.elharo.com/web/why-rest-failed/>

[3] *POST considered inconvenient*, Elliotte R. Harold, <http://cafe.elharo.com/web/post-considered-inconvenient/>

[4] *A proposal for a Common Execution Architecture*, Paul Harrison, <http://www.ivoa.net/Documents/latest/CEA.html>

---

<sup>5</sup> If the string form of ADQL is used, then the parameter can be sent with URL encoding and an HTML form can be used. If the XML form of ADQL is needed, then an ADQL document appears as the body of the posted request; in this case, a rich client is needed.