*International*

*Virtual*

*Observatory*

*Alliance*

# REST Basic Profile

# Version 1.0

## *IVOA Working Draft 2011 October 16*

**This version:**
http://www.ivoa.net/Documents/WD/GWS/VOREST-20111116.doc
**Latest version:**
http://www.ivoa.net/Documents/latest/VOREST.html
**Previous version(s):**
…

**Author(s):**
André Schaaff, Norman Gray, Pat Dowler
IVOA Grid and Web Services Working Group

## Abstract

In the early years of the VO, the SOAP Web Service paradigm was an important element of the IVOA Architecture. Developments around these services are more and more complex with an increasing number of standards (WS-* …). REST [3] is not a standard but a formalization of the URL use and it is easy to implement it. A service is RESTful if it follows a set of rules (which are not defined in a standard document). As there is no standard we think that it is necessary to define a minimal guideline about the "RESTfullness" in the VO context.

## Status of This Document

This is an IVOA Working Draft. The first release of this document was 2008 May 05.

*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

This document derives from discussions among the Grid and Web Services working group of the IVOA in Beijing (May 2007), Cambridge (September 2007), Trieste (May 2008), Strasbourg (May 2009) and Nara (December 2010).

## Contents

## 1 Introduction

Roy Fielding introduced the notion of Representational State Transfer (REST) in his 2000 PhD thesis [3]. Fielding's thesis is both readable and compact, and gives a very good account of the underlying motivation. Richardson and Ruby [REF] builds on this by elaborating the notion of the Resource-Oriented Architecture, and both more prescriptive and, possibly consequently, more practical than Fielding.

Fielding refers to an “architectural style” which is best matched (Fielding argues) to the features of the World Wide Web and its components and deployments. Fielding defines 'architectural style' as follows:

An architectural style is a coordinated set of architectural constraints that restricts the roles/features of architectural elements [ie components, connectors, and data] and the allowed relationships among those elements within any architecture that conforms to that style.

Fielding identifies and describes a number of architectural styles in use in the context of network-oriented programming, including several that readers of this present document will be familiar with. He identifies properties -- such as efficiency, simplicity or extensibility -- which the styles do or do not exhibit, and uses these as the basis for an analytic classification. For example, he describes the 'uniform pipe and filters' style adopted by Unix, which supports an ecology of programs which expect text-based input, and transform it to text-based output, in a way which has some efficiency drawbacks (everything must be serialized to and deserialized from text), but which is uniform and intelligible enough that applications can easily be inserted into the ecosystem, and composed and understood naturally.

Fielding then identifies the architectural problem represented by the Web (represented by its anarchism, simplicity and the requirement for extensibility), and identifies the architectural style, which has implicitly evolved in response, which is represented by technologies such as DNS, URIs and HTTP. His central claim, in three explicit theses, is that this implicit style can usefully be made explicit, that, with this done, the style can be self-consciously improved into a new 'hybrid style', and (a more technocratic point) that this can be used to guide the development of new or updated web standards. The resulting 'hybrid style' is Fielding's REST Architectural Style.

Before describing the style, it is worth noting that this is not presented as in any sense a _standard_, with RFC 2119-style 'shoulds' and 'musts', or reference implementations. If we identify a building as being 'in a baroque style' or a chair as 'regency', we are not claiming that the building or chair has been _validated_ as having ticked a set of boxes, but instead asserting that it would fit naturally alongside other things designed in the same style. The assertion behind Fielding's thesis, and behind this present IVOA document, is that if services are designed in a way which respects the constraints of the REST style, then they will fill naturally and usefully together, both within the IVOA and within the larger World Wide Web.

This IVOA document has two goals:

- By summarizing Fielding's work, we hope to produce or document some consensus, within the IVOA, of what the term 'RESTful' means, since we are aware that it has been used in a variety of senses which are not always compatible. This is the goal of section 2 of the document, below.
- We also have some prescriptive intent, in that we hope to create at least the expectation that new IVOA protocols should (in an RFC 2119 sense) be RESTful in the sense described in this document. This is the goal of section 3, below. This section is deliberately rather terse: we hope that the rationale for our prescriptions here is implicit in the longer discussion of section 2.

In section 4, we briefly discuss some of the tools and frameworks available to support a RESTful service, or the client of one. In the authors' experience, such frameworks are much less valuable than in the case of, for example, SOAP: it is a consequence of the REST 'uniform interface' (see below, section XXX) that a RESTful service is often reasonably easy to implement and to use by dealing with the REST layer in an ad hoc manner.

**Audience**: The intended audience for this document includes:

- Designers of IVOA standards specifying RESTful services.
- Implementers of services corresponding to these standards, and
- Implementers of clients using these services, who wish to understand some of the rationale behind the service design.

We presume that the readers of this document are familiar, at least in outline, with the use of URIs and the HTTP protocol, and with the RFCs specifying them. Given this background, we expect the following account of the REST style to be standalone; despite this, we aim to keep this document readably short, and if we have been too compact, we encourage readers to examine the main references on the subject for further discussion.

# 2 REST and the Resource Oriented Architecture

The following is intended as a brief summary. See Fielding's chapter 5 for fuller discussion, or references such as [Leonard Richardson and Sam Ruby, RESTful Web Services, O'Reilly 2007].

A service written in a RESTful style has the following features:

- Client-server: there is a separation of concern which, broadly speaking, leaves processing and data storage on the server, and user interfaces on the client side. This decoupling supports portability and scalability.
- Statelessness: each request from the client to the server must contain all of the information necessary to process the request -- any session state is managed by the client, which may use it to change the detail of future requests. This can occasionally complicate the client implementation, and potentially makes requests bigger than they might otherwise be, but this is repaid by simplifying the server implementation, in a way which increases the reliability and scalability of the server.
- Cacheability: this is seen as a virtue, where cacheing is possible this is made explicit, and servers, clients and (importantly) any network intermediaries are encouraged to rely on caches if they are available. This increases processing and network efficiency.
- Uniform interface: the REST style crucially encourages a uniform interface to services; we discuss this at greater length below. This partly decouples clients and services, which makes it possible for them to evolve separately, though at the cost (shared with the unix pipes style for example) that it is somewhat less efficient to convert everything to a common format. These inefficiencies are probably most acute for fine-grained hypermedia transfer, and it is probably for this reason that the AJAX style has emerged in the last decade or so, for those aspects of a client-server interaction which require fine-grained interaction.
- Other constraints: Fielding discusses two further constraints, covering layering and movable code, which are of less immediate interest to us in our present context.

**Resource Oriented Architecture -- the uniform interface**

While the advantages of a client-server separation, statelessness and cacheability are well-known and broadly uncontroversial, it is the notion of the uniform interface that is most distinctive about the REST style and, we think, the most misunderstood. This is also the point at which 'the REST architectural style', as a set of rather abstract design criteria, is instantiated in a particular set of design prescriptions. The best-known of the various possible instantiations of the style is the 'Resource-Oriented Architecture' associated with Richardson and Ruby [REF], although they were not the first to use this term. When we refer to

'REST', below, we are generally referring to Fielding's overall style; when we refer to ROA, it is to Richardson and Ruby's specialisation.

Fielding defines the 'uniform interface' as follows:

REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self- descriptive messages; and, hypermedia as the engine of application state. [Fielding00, Sect 5.1.5]

Taking this further, a Resource Oriented Architecture (ROA) [9; XXX more precise ref?] can be reduced to four concepts (resources, their names (URIs), their representation, the links between them) and four properties (addressability, statelessness, connectedness, uniform interface).

Within REST, the key notion is that of the *resource*, which is an abstract notion which might correspond to (i) a particular bug, (ii) the collection of all bugs, or (iii) the collection of today's bugs. Note that resources (such as (ii) or (iii)) can contain other resources (i), that some resource (i) always refer to the same thing, others (ii) to a changing collection of things, and others (iii) will refer to a different thing today from what they referred to yesterday.

*Resources* have *resource identifiers* (URIs) and *representations* which communicate the state or data of a resource to a client, in the form of HTML documents, images, or XML or JSON data. These in turn have *representation metadata* (MIME types, or dates); the resources also have *resource metadata* (alternates) and the retrieval of them involves *control data* (HTTP method names, or cache control). It is clear that there is a very strong association between REST and the HTTP protocol: this is because the HTTP protocol is fundamental to the web, is the protocol which generates the architecture that inspired REST, and when the protocol was revised in 1999 (RFC 2616) it was with Fielding's leading participation. Despite this, the (generic) REST architecture is not restricted to HTTP.

## The ROA and HTTP

The Resource-Oriented Architecture (ROA), in contrast, is significantly more closely bound to HTTP (this is not a major limitation, since its features have reasonably obvious analogies in other protocols).

## HTTP and statelessness

With HTTP, statelessness is to a large extent built in to the protocol. Since this is inconvenient to many web applications, a few techniques have emerged to support statefulness, including the use of cookies or nonce URLs which encode -- opaquely or transparently -- some server-managed state. In addition, it is well known -- indeed notorious -- that services such as Google or Facebook store very large amounts of state on their servers, and that almost any server will have state in the form of growing access logs.

Neither of these is a counterexample to the assertion that HTTP is stateless. In the first case, the state is being managed at the application level rather than the protocol level, and with the cooperation of the client who, in the case of cookies, effectively makes different queries (at the level of 'control data') depending on the state it has been asked to curate, and in the case of nonce URLs makes stateless queries to transient URLs. Similarly, the large quantity of server-side state that may accumulate in a Facebook-like service does not exist at the protocol level, and the client is neither responsible for, nor has control over, this state.

One way of thinking about the resources within an ROA is that each of the named resources (named by URIs, remember) *names a piece of state* of the underlying application, such as the existence, the state, or the output of a batch job.

**HTTP verbs and the uniform interface**

One aspect of the REST 'uniform interface' finds expression in ROA in the uniform usage of the small number of HTTP methods, which can be regarded as the 'verbs' of the HTTP protocol. There are eight methods defined (and further ones in HTTP-like protocols such as WebDAV), of which the four key ones are PUT, GET, POST and DELETE, which together broadly implement the four CRUD operations of database theory.

This uniformity is in contrast to the flexibility of an interface which was inspired by an object-oriented pattern. Within such a paradigm, there will typically be rather few objects, corresponding to the main conceptual entities in a model of the system, but a rich array of methods, each of which must be documented. In contrast, in an ROA, there will typically be a larger number of objects/resources, each of which is named by a URI, and each of which has exactly the same interface, namely the four verbs GET, POST, PUT and DELETE. It may not make sense, or it may be forbidden, for a client to attempt to DELETE a resource, but it is not a protocol or API error to attempt to do so. This means that the interface does not have to be documented at all at the level of 'methods', and that the design and documentation of the service finds expression in the tree of resources. We should not that there is no in-principle reason why an ROA architecture exposes resources structured into a tree, with longer paths naming sub-resources of shorter ones, but this feature is common, easy for the client (programmer) to understand, and easier to implement.

**Representation formats**

A resource may have multiple representations; for example, a resource naming a job in a batch queue may have an HTML representation for display, plus VOTable and JSON representations of the same information for the convenience of different types of client. The client can distinguish these representations using the MIME type in the HTTP Content-Type response header, and control which one is sent by using the Accept request header.

Although this multiplicity of representations is not particularly profound, it turns out to be characteristic of RESTful applications that they are 'willing' to produce a variety of response formats. There is no particular consensus on exactly how this is managed: some applications will immediately return the content which was requested in the Accept header, while others will instead respond with a redirection to another URL which returns only that representation. Either is valid.

**The uniform interface and 'safe and idempotent methods'**

One aspect of the uniform interface which ties in with the cacheability of HTTP is the HTTP specification's identification of the GET and HEAD methods as 'safe', and the GET, HEAD, PUT and DELETE methods as 'idempotent' [REF RFC 2616, Sect 9.1.2]

The 'safe' methods do not cause any side effect on the server which the client can be held responsible for -- they are intended purely for retrieval, and any side-effects which do happen (such as writing a log entry) are the server's responsibility. This means that _any_ GET request can have its response cached (ignoring for the moment any explicit cache-control directives in the response). The 'idempotent' methods are such that the effect of multiple identical requests is the same as one request.

Together, these two properties mean that if a client were to make a GET request via a cache which had a copy of a resource (and again putting aside cache-control state), the cache would be acting correctly whether it made zero, one or more requests to the ultimate server (what REST refers to as the 'origin server'). This means in turn that GET cannot safely be used to mutate state on a server, and a design which depends on one client GET request turning into precisely one GET request at the server, is a design which is fundamentally broken.

**The uniform interface, hypermedia, and connectedness**

The aspect of the REST style's web service 'uniform interface' that is most poorly understood, in our experience, is its focus on hypermedia. This is odd, since it is the notion of the link which is most clearly understood by most people, when thinking about the traditional web.

Fielding (section 5.3.3) refers to this as a 'data view of an architecture', which emphasises the notion that the client's view of a server-side application is based on the data in the representation the client receives. Fielding also refers to this with the slogan 'hypermedia as the engine of application state', which has the unlovely acronym 'HATEOAS'. Richardson and Ruby describe this with the rather clearer term 'connectedness' [R&R, ch4]

When we look at an HTML web page, we can very quickly see where to go next, since the web page provides links embedded within it -- as Richardson and Ruby stress, we don't have to go from page to page by guessing or deducing related URLs, and typing them into the address bar. We would find this intolerable in a web page; we should find it equally intolerable in a machine-readable resource representation.

What this means in practice is that, when we are choosing a representation to return as part of a web service we are designing, we should ensure that we include machine-readable links to other 'interesting' resources, as far as possible. Often, this representation will comprise one or other XML vocabulary, or perhaps JSON, and we can take advantage of structures within these to point to parent resources, perhaps, or the next in a sequence. A client which knows how to identify such links can take advantage of them (perhaps presenting them to a user) with no further documentation.

Of course, we may not have any choice about the type of representation to return -- for example, it might be that the only reasonable representation is a VOTable, which does not have natural places for such links to appear -- and in this case there is little we can do, beyond adding such links where possible, and perhaps agitating for suitable support in future representation versions.

The issue of connectedness is linked to the question of whether a service should support 'semantic' or 'opaque' URLs to name resources. 'Semantic' URLs are those which reveal the internal or conceptual structure of the resources within the URL, so that, for example, the URL for a batch queue is a prefix of the URLs for each of the jobs in the queue. In contrast 'opaque' URLs do not reveal this structure, or do not do so reliably. If one is being purist about the REST style, then one should probably prefer opaque to semantic URLs, as a service designer, on the grounds that semantic URLs are brittle (you can't change their structure without breaking applications), are burdensome (the structure must be documented, if clients are expected to construct them), and should be unnecessary (since clients would ideally be led from URL to URL by well-connected representations, as discussed below in Sect. XXX). In practice, it is probably more effort to create unstructured URLs, and these are generally easier to understand and debug for the user. Since, again ideally, a service will produce well-connected representations, it should rarely be necessary for a client to construct URLs.

# 3 RESTful services within the IVOA

For a service to be accepted as 'RESTful' as an IVOA Recommendation, it should exhibit the following properties.

*Yes, these are very prescriptive: we expect these to be adjusted in subsequent versions of this document*

**All resources are named by suitably expressive URLs.** The URLs should describe a large enough range of the service's concepts that each resource can be reasonably read or manipulated using the (HTTP) uniform interface's CRUD methods.

**Resources should support the HTTP CRUD methods where appropriate.** Of course, it does not follow that every resource can have a useful reaction to GET/POST/PUT/DELETE, but if the resource is to be read or manipulated, it should be through these methods.

**Services should support multiple representations where appropriate.** Services will have human users as well as machine ones, and it makes sense to give these two categories of users different representations.

**Services should include connections within the representations they return, if this is syntactically feasible.** Ideally, it should be possible for a client to go from one resource to another, with minimal documentation required.

**More?**

# 4 Representational State Transfer

## 4.1 Quick definition of REST and RESTful

In the REST approach, it may be sufficient to know the URI to access to a resource.

Examples:

1) http://www.example.com/sky/m31/pictures

2) http://www.example.com/sky/m31/picture/1

In these examples it is possible to access to the information through a simple URL without the use of a specific tool (for C#, Java, Perl …) on the client side, the client has just to read simply the URL.
In example 1) the URL returns the number of available pictures for m31 and in example 2) the URL returns the first picture.

Main feature of a REST service, from [9]:

- Architectural style of the Web.
- Resources are addressable (URIs).
- Interact with representations of resources.
- State is maintained within a resource representation.
- Small set of methods that can be applied to any resource (HTTP methods).
- Scalable, low cost of coordination.

To be RESTful a service must be compliant with the following principles:

- Addressability
- Stateless
- Connectivity
- Uniform interface

## *4.2 Quick comparison with SOAP services*

If we take again the two previous examples of REST [3] URLs,

1) http://www.example.com/sky/m31/pictures

2) http://www.example.com/sky/m31/picture/1

In SOAP we will have to define something like "int getPictures (String object)" for 1) and to use for example SOAP with attachment mechanism or to return the URL of the image for 2).

SOAP Web Service engines are also evolving by implementing the REST [3] alternative. For example, in Axis 2 it is possible for a client to specify that he wants to access the service following the REST [3] paradigm.

## *4.3 How to describe a service?*

WSDL (Web Services Description Language) [7] has been created to describe SOAP services. These services are self-described by interrogation of the service

endpoint URL with an extension like “?wsdl”. It is difficult to use a SOAP Web Service without this description like when you try to use a Java API without the corresponding Javadoc. For REST [3], there is no standard way to provide a self-description. It is possible to use for example WADL (Web Application Description Language) [8] or also WSDL 2.0 but the service provider has to write it.
RESTful services have simpler interfaces and the description is not as important as for SOAP Web Services. But in the case of automatic creation or use of the services by tools it is necessary to provide a description of the services. If a WADL [8] description is available it is then possible to generate for example the Java client code to query the service. See [14] for a quick WADL – WSDL 2.0 comparison.

## 4.4 WADL

WADL (Web Application Description Language) [8] is a draft specification for an analogue to the WSDL language, specialized to RESTful interfaces.

The WADL distribution includes an XML schema for WADL files, schema documentation, and some tools which generate documentation and client-side Java stubs from an input WADL file. The specification is an advanced draft, but it is not clear where further standardization will take place, nor when. As with WSDL, the goal with WADL is to document an interface in a machine-readable form. For example, the following WADL file describes a simple interface which allows clients to GET HTML representations of resources:

```
<application xmlns="http://research.sun.com/wadl/2006/10">
  <resources base='http://example.org/resource'>
    <resource path='{resourceName}'>
      <doc>This resource is one of a set of resources</doc>
      <param name='resourceName' style='template'>
        <doc>The name of the resource being described</doc>
      </param>
      <method name='GET'>
        <response>
          <representation status='200' mediaType='text/html'>
            <doc>Returns a description of the resource</doc>
          </representation>
          <fault status='404' mediaType='text/html'>
            <doc>If the document is not found, return an explanation</doc>
          </fault>
        </response>
      </method>
    </resource>
  </resources>
</application>
```

This describes a set of resources http://example.org/resource/{resourceName} for different values of the 'variable' {resourceName}. A GET request may produce one of two responses, namely a text/html response with a 200 status, or another text/html response, describing an error, with a 404 status.

In the case of WS-* services, a WSDL description is almost essential; there are so many technicalities involved in making a WSDL call, that a client application author is almost certain to make mistakes if they attempt to implement the client interface by hand. Also, the expectation of WS-* services is that the contents of the request and response are representations of program objects, which must be serialized into a request, and desterilized from a response, again introducing many opportunities for error.

The REST paradigm, however, avoids the fragility of WS-* services, by insisting that the contents of HTTP responses (and HTTP requests where appropriate) be _representations_ of the corresponding resources, encoded in one or other MIME type included in the HTTP request or response. This implies that the problems of serialization and deserialization are external to the protocol; this makes the interface easier to describe, with the strong advantage that the separation between the interface and the representations it carries is more clearly distinct.

Although this appears to place more of a burden on the client application, this is not the case in practice: since an application generally has to ingest representations of resources anyway, from files or other URLs, it is usually capable of ingesting representations from a RESTful interface without difficulty. What this means in turn is that a RESTful interface is generally rather straightforward to implement on the client side, with a lot less 'glue' which is specific to the interface.

In consequence, there is a much lesser need for the sort of generated code stubs which are a crucial output of WSDL tools. This is fortunate, since the code-generation tools distributed with the WADL standard seem immature, sometimes failing on valid input, and appearing to work naturally only for that subset of services which have a predominantly keyword-value GET interface.

In the experience of one of the present authors ([12], NG), a WADL file is a useful component of a RESTful service, even without any generated client code. The WADL file provides a usefully explicit specification of the service's interface, which was used to generate human-readable documentation and, using another custom XSL transformation, to generate code which verified that the service's test cases exercised the entire interface, and did not violate it at any point. A variant of this checking code could have been (but was not in fact) included within the server to guarantee that the service's responses matched its interface promises.

Thus on this and similar cases, the WADL file was useful enough, internal to code-base, to justify its use, and offering the WADL file to users of the service was an added bonus.

Different works about the generation of a WADL file for a service and about the code generation in different language from a WADL description (Java, Python, Ruby…) are on going. See and try for example [13].

# 5 REST oriented frameworks and tools (not exhaustive)

## 5.1 Ruby on Rails

Rails is a framework written in Ruby and dedicated to Web developments.
See [4] for more details.

## 5.2 Restlet

Restlet is a framework for the Java platform providing native REST [3] support.
See [5] for more details.

## 5.3 Django

Django  is an open source framework for the Python expected to provide a native REST [3] support in the coming months.
See [6] for more details.

## 5.4 NetKernel

NetKernel is an implementation of a resource-oriented computing abstraction. It can be thought of as an Internet-like operating system running on a microkernel within a single computer. It is available with 2 kinds of license: open source and commercial.
See [10 for more details.

## 5.5 Apache CXF

Apache CXF is an open source services framework designed to build and develop services using front-end programming APIs. Protocols such as SOAP, XML/HTTP, RESTful HTTP, or CORBA and of transports such as HTTP, JMS or JBI are possible.

See [11] for more details.

### 5.6 Comments

REST development or compliant frameworks are evolving quickly so we think that it is difficult to recommend a restricted set of them.
As an example of Restlet use, it is possible to refere to one of the reference implementation of VOSpace 2.0 which is based on REST.

## 6 Restful services in the VO

### 6.1 Interoperability problems

As said in a previous part of this document, REST [3] is not a standard. The SOAP approach which was a key element in the IVOA architecture is a standard but is also crushed under a huge stack of related standards (WS-*).
As VO services are not just designed for humans but also to be queried by another tools, it would be very efficient to have a "standardized" description of the REST [3] services provided in the frame of the IVOA.
It could also be useful to have a tool to check if the service follows a minimal set of rules.

### 6.2 Recommendations

Since REST is not a formal standard, but instead a set of good practices, services cannot be required to 'conform' to REST. We RECOMMEND, however, that future services should conform to these good practices wherever possible, and that service authors should seek feedback on their interface design from the GWS WG, with a view to ensuring that the service conforms to the spirit of these practices as much as possible.

Although WADL is not, or not yet, a standard, we cannot require conformance to that, either. However the practical advantages to a service implementation of having a machine-readable interface specification (as described in section 3.4 above), and the interoperability advantages of having a public commitment to an interface, are substantial enough that we RECOMMEND that services publish a WADL file.

### 6.3 Work to done

The WADL spec (section 5) suggests that WADL documents should be served using the application/vnd.sun.wadl+xml MIME type, and there are suggestions elsewhere that they should be available at a http://example.org/service?wadl URL (though this is not mentioned in the WADL spec). Should we echo this, and generally be more prescriptive here?

## 7 Conclusion

Compared to SOAP Web Services REST [3] is a light way to provide a Web service following just a reduced set of rules. But it is perhaps necessary to define clearly the basis of what an interoperable RESTful VO service must be. REST [3] is more human oriented than SOAP but it defines no standard concerning the description of the service which is important in the case of a dynamic use by other services. At least we recommend to provide the description of a REST service through a formalism like WADL.

## 8 Changes since last version

Updates and corrections.
Rename the document title from “REST in the VO” to REST Basic Profile.

## 1 Appendix A: “Appendix Title”

…

## 2 References

[1] R. Hanisch, *Resource Metadata for the Virtual Observatory* , http://www.ivoa.net/Documents/latest/RM.html
[2] R. Hanisch, M. Dolensky, M. Leoni, *Document Standards Management: Guidelines and Procedure* , http://www.ivoa.net/Documents/latest/DocStdProc.html
[3] R. Fielding, http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
[4] Ruby on Rails, http://www.rubyonrails.org/
[5] Restlet, http://www.restlet.org/
[6] Django, http://www.djangoproject.com/
[7] WSDL, http://www.w3.org/TR/2007/REC-wsdl20-primer-20070626/
[8] WADL, http://www.w3.org/Submission/wadl/
[9] RESTful Web Services, L. Richardson and S. Ruby, O’Reilly
[10] NetKernel, www.1060.org
[11] Apache CXF, http://cxf.apache.org
[12] SKUA project, http://myskua.org/
[13] REST Describe (and compile), http://tomayac.de/rest-describe/latest/RestDescribe.html
[14] WSDL 2.0 vs WADL, J. Normand, http://www.ivoa.net/cgi-bin/twiki/bin/view/IVOA/InterOpMay2009GridAndWebServices