GWD-R
OCCI-WG

Thijs Metsch, Platform Computing
Andy Edmonds, Intel
October 7, 2010
Updated: December 6, 2010

Open Cloud Computing Interface - RESTful HTTP Rendering

Abstract

This document, part of a document series, produced by the OCCI working group within the Open Grid Forum (OGF), provides a high-level definition of a Protocol and API. The document is based upon previously gathered requirements and focuses on the scope of important capabilities required to support modern service offerings.

(Andy: need to set the official version of OCCI - is it 1.1, 0.2, 1.0??)

(Andy: Kind sub-types are mentioned in the doc as are resource instances. Need to align this terminology with the core and infrastructure documents e.g. a REST resource is the equivalent of an 'entity sub-type' instance. Could be dealt with by glossary.)

Ralf: Yes, Kind need be replaced with Entity in serveral places. "resource instance" is ok to use (when referring to Entity sub-type instance), it is in the glossary already

(Andy: Filtering mechanism should be MUST)

(Andy: little reference to collections)

(Andy: there are references to service provider, service etc. This should be aligned to core and infrastructure. There is is known as an 'OCCI implementation')

(Andy: header should be referred to as 'HTTP header' throughout)

Contents

1 Introduction

The Open Cloud Computing Interface (OCCI) is a RESTful Protocol and API for all kinds of Management tasks. OCCI was originally initiated to create a remote management API for IaaS[1] model based Services, allowing for the development of interoperable tools for common tasks including deployment, autonomic scaling and monitoring. It has since evolved into an flexible API with a strong focus on interoperability while still offering a high degree of extensibility. The current release of the Open Cloud Computing Interface is suitable to serve many other models in addition to IaaS, including e.g. PaaS and SaaS.

In order to be modular and extensible the current OCCI specification is released as a suite of complimentary documents which together form the complete specification. The documents are divided into three categories consisting of the OCCI Core, the OCCI Renderings and the OCCI Extensions.

- The OCCI Core specification consist of a single document defining the OCCI Core Model. The OCCI Core Model can be interacted with *renderings* (including associated behaviours) and expanded through *extensions*.
- The OCCI Rendering specifications consist of multiple documents each describing a particular rendering of the OCCI Core Model. Multiple renderings can interact with the same instance of the OCCI Core Model and will automatically support any additions to the model which follow the extension rules defined in OCCI Core.
- The OCCI Extension specifications consist of multiple documents each describing a particular extension of the OCCI Core Model. The extension documents describe additions to the OCCI Core Model defined within the OCCI specification suite.

The current specification consist of three documents. Future releases of OCCI may include additional rendering and extension specifications. The documents of the current OCCI specification suite are:

OCCI Core describes the formal definition of the the OCCI Core Model [?].

OCCI HTTP Rendering defines how to interact with the OCCI Core Model using the RESTful OCCI API [?]. The document defines how the OCCI Core Model can be communicated and thus serialised using the HTTP protocol.

OCCI Infrastructure contains the definition of the OCCI Infrastructure extension for the IaaS domain [?]. The document defines additional resource types, their attributes and the actions that can be taken on each resource type.

2 Notational Conventions

All these parts and the information within are mandatory for implementors (unless otherwise specified). The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [?].

All examples in this document use one of the following three Categories. An example namespace is also given. Syntax and Semantics is explained in the remaining sections of the document. These examples do not strive to be complete but to show the functionalities OCCI has:

Ralf: Instead of "Categories" say "HTTP Category Headers representing two Kind instances and one Mixin instance" or similar.

That would make this text unreadable – -1

[1] Infrastructure as a Service

```
Category: compute;
          scheme=http://schemas.ogf.org/occi/infrastructure;
          location=/compute
                                (This is an compute kind)

Category: storage;
          scheme=http://schemas.ogf.org/occi/infrastructure;
          location=/storage
                                (This is an storage kind)

Category: my_stuff;
          scheme=http://example.com/occi/my_stuff;
          location=/my_stuff
                                (This is a mixin of user1)

The following namespace hierarchy is used in the examples:

http://example.com/-/
http://example.com/vms/user1/vm1
http://example.com/vms/user1/vm2
http://example.com/vms/user2/vm1
http://example.com/disks/user1/disk1
http://example.com/disks/user2/disk1
http://example.com/compute/
http://example.com/storage/
http://example.com/my_stuff/
```

By default all examples given in this document created using *text/plain* as Content-Type and Accept HTTP header.

The following notations are used when referring to parts or complete URIs:

```
http://www.example.com:8080/foo/bar;action=stop
<   >  <      Authority     >< Path >< Fragment >
  ^
  Scheme
```

3 A RESTful HTTP Rendering for OCCI

Ralf: The intro text in below gives a nice description of OCCI and REST. However I am missing the relation to OCCI Core here. I.e. that OCCI HTTP Rendering in fact is just *one* rendering of possibly many etc. Adding an some text describing OCCI Core vs OCCI renderings first and then jumping in to the HTTP REST specifics should solve it nicely though.

The HTTP Protocol is the underlying core fabric of OCCI and OCCI uses all the features of the HTTP and underlying protocols offer. OCCI also builds upon the Resource Oriented Architecture (ROA). ROA's use Representation State Transfer (REST) **(Andy: use references of Feldings thesis and the O'Reilly book on REST)** to cater for client and service interactions. Interaction with the system is by inspection and modification of a set of related resources and their states, be it on the complete state or a sub-set. Resources MUST be uniquely identified. HTTP is an ideal protocol to use in ROA systems as it provides the means to uniquely identify individual resources through URLs as well as operating upon them with a set of general-purpose methods known as HTTP verbs. These HTTP verbs map loosely to the resource related operations of Create (POST), Retrieve (GET), Update (POST/PUT) and Delete (DELETE).

The following section describe the general behaviour for all HTTP based renderings. Later sections will describe the syntax and semantic of how to render the OCCI Core model with different Content-Types.

Each Kind sub-type instance **(Andy: This now should be Entity sub-type)** within an OCCI system must be uniquely identified by an URI. The structure of these URIs is opaque and the system should not assume a static, pre-determined scheme for their structure (Example: *http://example.com/vms/user1/vm1*).

3.1 Behaviour of the HTTP Verbs

As OCCI adopts a ROA, REST-based architecture and uses HTTP as the foundation protocol the means of interaction with all RESTful resource instances is through the four main HTTP verbs **(Andy: We should mention the existence of other verbs too (HEAD, OPTIONS))**. OCCI service implementations MUST, at a minimum, support these verbs as shown in Table 1:

Table 1. HTTP Verb Behaviour **(Andy: Can this table be completely ruled? Is hard to read left to right)** Ralf: Done. Still a bit hard to read though.

Type	GET	POST (create)	POST (action)	PUT (create)	PUT (update)	DELETE
resource instance **(Andy: this needs to be aligned to core and infrastructure terminology)**	Rendering of this instance	Create a new instance	Trigger action **(Andy: attributes/ parameters?)**	Create an instance at the given path	Update an instance at an given path	Delete this resource instance
Path in the namespace hierarchy ending with /	Listing of all instances below this namespace	Create a new instance	N/A	N/A	N/A	Delete all the entries below this namespace hierarchy
Location of an Mixin or Kind	URI-Listing containing Paths to all entities belonging to this Mixin or Kind	N/A	Trigger action (defined for this kind or mixin) on all resource entities belonging to this Mixin or Kind **(Andy: attributes/parameters?)**	Add an resource instance to a mixin	N/A	Delete an instance given in the request from a mixin
Query Interface **(Andy: new reader 'what's this?' perhaps introduce or reference section)**	Listing of all registered categories	N/A	N/A	Add a user defined mixin	N/A	remove a user-defined mixin (defined in the request)

3.2 A RESTful Rendering of OCCI

The following sections and paragraphs describe how the OCCI model MUST be implemented by service providers. Operations which are not defined are out of scope for this specification and MAY be implemented by the service provider. This is the minimal set to ensure interoperability.

3.2.1 Namespace Hierarchy and Location

Ralf: I think we need a section explicitly describing the mapping of OCCI Core to OCCI HTTP rendering. After all, an OCCI rendering describes "how to interact with the OCCI Core model". This section should describe:

- The flat names-pace used in Core, i.e. just a bunch of IDs (Entity.id).
- The hierarchical names-pace of RESTful HTTP (very brief).

- How the hierarchical HTTP names-pace is mapped into the OCCI Core names-pace. (This is the important stuff.)

Ralf: Names-pace and "names-pace hierarchy" need to be defined, but the above described section should handle that.

The namespace and the hierarchy are free definable by the Service Provider. Although the Service Provider MUST implement the location path feature**(Andy: reference to this section needed)**, which is required by OCCI for discovery capabilities and operations on Mixins and Kinds. Location paths tell the client where all resource instance of one Kind or Mixin can be found regardless of the hierarchy the service provider defines. These paths are discoverable by the client through the Query interface **(Andy: section ref needed)**.

These location paths can be part of the namespace or rendered alongside. The following example shows how the locations paths are rendered alongside the namespace hierarchy:

```
Category: compute;
          scheme=http://schemas.ogf.org/occi/infrastructure;
          location=/compute
                    (This is an compute kind)
Category: storage;
          scheme=http://schemas.ogf.org/occi/infrastructure;
          location=/storage
                    (This is an storage kind)

Category: my_stuff;
          scheme=http://example.com/occi/my_stuff;
          location=/my_stuff
                    (This is a mixin of user1)

The following namespace hierarchy is used in the examples:

http://example.com/-/
http://example.com/vms/user1/vm1
http://example.com/vms/user1/vm2
http://example.com/vms/user2/vm1
http://example.com/disks/user1/disk1
http://example.com/disks/user2/disk1
http://example.com/compute/
http://example.com/storage/
http://example.com/my_stuff/
```

Location paths can also be part of the namespace hierarchy:[2]

(Andy: what's the difference with the two examples? It's not obvious)

```
Category: compute;
          scheme=http://schemas.ogf.org/occi/infrastructure;
          location=/vms
                     (This is an compute kind)

Category: storage;
          scheme=http://schemas.ogf.org/occi/infrastructure;
          location=/disks
                     (This is an storage kind)

Category: my_stuff;
```

[2]/vms/user1/vm1 (= OCCI base type ID) is a resource instance below the namespace path /vms/user1/.

```
          scheme=http://example.com/occi/my_stuff;
          location=/my_stuff
                     (This is a mixin of user1)

The following namespace hierarchy is used in the examples:

http://example.com/-/
http://example.com/vms/user1/vm1
http://example.com/vms/user1/vm2
http://example.com/vms/user2/vm1
http://example.com/disks/user1/disk1
http://example.com/disks/user2/disk1
http://example.com/my_stuff/
```

3.2.2 Various Operations and their Prerequisites and Behaviours

3.2.2.1 Operations on Resource Instances The following operations MUST be implemented by the service provider for operations on resource instances. The resource instance is uniquely identified **(Andy: say how it maps to in core Entity::id)** by an URI (For example: *http://example.com/vms/user1/vm1*).The path MUST not end with an '/' **(Andy: say why, what does it mean if '/' is appended)**.

Creating a Resource Instance A request to create a resource instance MUST contain at least one Category definition Ralf: Say "Category Header" instead of "Category definition". Otherwise it is easy to mix-up this with the OCCI Core model Category type. Even when using text/plain rendering it is "Category Headers" that we put into the body. It least from a syntax point of view. which is (or relates to) a Kind definition. If multiple Categories are defined the first one which is (or relates to) to a Kind MUST be used for defining the type of the resource instance **(Andy: Is this defined in core? If not should it in some way? This mechanism seems somewhat arbitrary.)**. Ralf: I would rather say something like: "If multiple Category Headers are supplied in a request and they refer to more than one unique Kind instance, ...". That would be OK with Core. However, I would argue this particular case MUST return Bad Request. Optional information which might be provided by the client and if available MUST be used are Links and Attributes. Two ways can be used to create a new resource instance - HTTP POST or PUT **(Andy: HTTP 200 is mentioned here and makes no reference to 202 until much later in doc. Should mention/reference it here)**:

```
> POST / HTTP/1.1
> [...]
>
> Category:compute;scheme=http://schemas.ogf.org/occi/infrastructure
> X-OCCI-Attribute:occi.compute.cores=2 occi.compute.hostname=foobar
> [...]

< HTTP/1.1 200 OK
< [...]
< Location:http://example.com/vms/user1/vm1
```

The path on which this POST verb is executed can be any existing path in the hierarchy of the service providers namespace. The service must return the Location of the newly created resource instance.

Or HTTP PUT can be used. In this case the client ask the service provider to create a resource instance at a certain path in the namespace hierarchy.[3] **(Andy: PUT will fail if the identifier supplied as the last fragment already exists. Implementations must ensure unique identities for resources.)**

[3] If a Service Provider does not want the user to define the path of a resource instance it can return a Bad Request return code - See section 3.4.4.

```
> PUT /vms/user1/my_first_virtual_machine HTTP/1.1
> [...]
>
> Category:compute;scheme=http://schemas.ogf.org/occi/infrastructure
> X-OCCI-Attribute:occi.compute.cores=2 occi.compute.hostname=foobar
> [...]

< HTTP/1.1 200 OK
< [...]
```

The service will return an OK code.

Retrieving a Resource Instance For retrieval the HTTP GET verb is used. It MUST return at least the Category which defines the Kind of the resource instance. Links pointing to related Ralf: When referring to the "Link Header" say "Link Header" instead of just Link/Links. resource instances, other URI or Actions MUST be included if present. Those Links SHOULD only be the Actions which are currently applicable **(Andy: remember we should be able to use the general Link specification too, e.g. link to documentation.)**. The Attributes of the resource instance MUST be exposed to the client if available.

```
> GET /vms/user1/vm1 HTTP/1.1
> [...]

< HTTP/1.1 200 OK
< [...]
< Category:compute;scheme=http://schemas.ogf.org/occi/infrastructure
< Category:my_stuff;scheme=http://example.com/occi/my_stuff
< X-OCCI-Attribute:occi.compute.cores=2 occi.compute.hostname=foobar
< Link: [...]
```

Updating a Resource Instance Before updating a resource instance it is RECOMMENDED that the client first retrieves the resource instance. Updating is done using the HTTP PUT verb. Only the information (Links, Attributes or Mixin Categories), which are updated MUST be provided along with the request.[4] **(Andy: Is this advocating partial-updates? If so recommend that this is removed in favour of full updates. Partial-PUTs are not clear in REST systems.)** Ralf: I think it is. I argue we keep it that way, i.e. do support partial updates. You can always supply the whole attribute-set if you want a full update. We are not fully REST compliant in any case.

```
> PUT /vms/user1/vm1 HTTP/1.1
> [...]
>
> X-OCCI-Attribute: occi.compute.memory=4.0
> [...]

< HTTP/1.1 200 OK
< [...]
```

Deleting a Resource Instance A resource instance can be deleted using the HTTP DELETE verb. No other information SHOULD be added to the request. **(Andy: What are the effects on possibly linked resources or resources that have Links to the resource to be deleted?)**

```
> DELETE /vms/user1/vm1 HTTP/1.1
> [...]

< HTTP/1.1 200 OK
< [...]
```

[4]Changing the type of the resource instance MUST not be possible.

Triggering an Action on a Resource Instance To trigger an action on a resource instance the request MUST containing the Category **(Andy: The example does not show this.)** defining the Action. It MAY include attributes which are the parameters of the action. Actions are triggered using the HTTP POST verb and by adding a fragment to the URI. This fragment exposes the term of the Action. If an action is not available or a proper return code **(Andy: what code precisely?)** should be returned.

```
> POST /vms/user1/vm1;action=stop HTTP/1.1
> [...]
> X-OCCI-Attribute:method=poweroff

< HTTP/1.1 200 OK
< [...]
```

Ralf: I like the Action implementation. I.e. action=<term> in the URI and attributes as X-OCCI-Attribute headers. Only thing I want to change is to use `?action=...` instead of `;action=...` The '?' query separator is recommended by RFC3986.

3.2.2.2 Handling the Query Interface

The query interface MUST be implemented by all services supporting OCCI. It MUST be found at the path */-/* **(Andy: should be noted that /-/ is off the root of the implementation's domain e.g. http://foo.bar/-/)** Ralf: Yes, and `http://example.com/api/-/` should also be allowed . The following operations, listed below, MUST be implemented by the service.

Ralf: In the query interface section it gets a bit tricky talking about "Category Headers", "Kind instances", "Mixin instances" and "Category instances (used to identify Actions)". These are all different things. The "Category Header" is used to render Kind, Mixin and Category instances. Saying "Adding a Mixin category definition" need "Mixin category" to be defined somewhere. Saying "Adding a Mixin *instance*" would be more correct. Either way is probably ok as long as the terminology used is defined somewhere.

Retrieving All Registered Categories The HTTP verb GET must be used to retrieve all categories the service can handle. This allows the client to discover the capabilities of the Service provider. The result MUST contain all information about the Category (including Attributes and Actions assigned to this Category). **(Andy: content type? appears inline and not URI-List is the default - not clear.)**

```
> GET /-/ HTTP/1.1
> [...]

< HTTP/1.1 200 OK
< [...]
< Category:compute;scheme=http://schemas.ogf.org/occi/infrastructure; \
                  attributes=occi.compute.cores,occi...; \
                  rel=http://schemas.ogf.org/occi/core\#entity; \
                  actions=http://schemas.ogf.org/occi/infrastructure/compute/action#stop,...;
                  location=/compute
< Category:my_stuff;scheme=http://example.com/occi/my_stuff; \
                   location=/my_stuff
< Category: storage; scheme=http://schemas.ogf.org/occi/infrastructure;
<     attributes="...";
<     actions="...";
<     rel=http://schemas.ogf.org/occi/core\#entity;
<     location=/storage
<
```

(Andy: Useful if the backslash character in the verbatim sections is explained in the conventions section) Ralf: I would rather remove the backslashes altogether. They are not necessary. RFC2616 allow line-breaking HTTP Headers. I have changed the last Category Header above to show what it would look like.

Note: A Service provider SHOULD support a filtering mechanism. If a Category is provided in the request **(Andy: body or header?)** the server SHOULD only return the complete rendering of the provided Category.

Adding a Mixin category definition (Andy: this feature is new - I presume it's related to adding tags? Doesn't make sense for a client to add new 'functional' mixins) Ralf: It is for user-defined tags. Agreed that attributes/actions probably do not make much sense here. To add a Mixin to the service the HTTP PUT verb MUST be used. All possible information for the Mixin category must be defined. At least the Category term, scheme and location MUST be defined. Actions and Attributes are optional:

```
> GET /-/ HTTP/1.1
> [...]
> Category:my_stuff;scheme=http://example.com/occi/my_stuff; \
                    rel=http:/example.com/occi/something_else#mixin; \
                    attributes=...; \
                    actions=...; \
                    location=/my_stuff

< HTTP/1.1 200 OK
< [...]
```

The service might reject this request if it does not allow user-defined Categories to be created. Also on name collisions of the defined location path the service provider might reject this operation.

Removing a Mixin category definition (Andy: again, this is new but we need to state its purpose. Is it for user management of tags?) A user defined Mixin CAN be removed (if allowed) **(Andy: removed from what?)** by using the HTTP DELETE verb. The information about which Mixin should be deleted MUST be provided in the request:

```
> DELETE /-/ HTTP/1.1
> [...]
> Category:my_stuff;scheme=http://example.com/occi/my_stuff;

< HTTP/1.1 200 OK
< [...]
```

3.2.2.3 Operations on Mixins or Kinds

All the following operations CAN only be be done on a location path provided by category definition. It MUST end with an /.

Retrieving All Resource Instances Belonging to Mixin or Kind (Andy: Should refer to collections in core here)

The HTTP verb GET must be used to retrieve all resource instances. The service provider MUST return a URI-list containing all resource instances which belong to the requested Mixin or Kind:

```
> GET /compute/ HTTP/1.1
> [...]

< HTTP/1.1 200 OK
< Content-type: text/uri-list
< [...]
<
< http://example.com/vms/user1/vm1
< http://example.com/vms/user1/vm2
< http://example.com/vms/user2/vm1
```

Note: A Service provider SHOULD support a filtering mechanism. If a Category **(Andy: Should use Kind and Mixin here.)** is provided in the request the server SHOULD only return the resource instances belonging to the provided Category. The provided category definition SHOULD be different from the one Category definition which defined the location path used in the request.

Ralf: Core currently requires an OCCI client to be able to retrieve a subset of a collection. Maybe change the above to a MUST or introduce paging? **(Andy: +1 to Ralf's comment)**

Triggering Actions on All Instances of a Mixin or Kind Actions can be triggered on all resource instances of the same Mixin or Kind. The HTTP POST verb MUST be used. Also the Action MUST be defined by the category which defines the location path which is used in the request:

```
> POST /compute/;action=stop HTTP/1.1
> [...]
 X-OCCI-Attribute:method=poweroff

< HTTP/1.1 200 OK
< [...]
```

Adding a Resource Instance to a Mixin (Andy: Might be better phrased as 'adding a mixin to a resource instance') One or multiple resource instances can be associated with a Mixin using the HTTP PUT verb. The URIs which uniquely defined the resource instance MUST be provided in the request:

```
> PUT /my_stuff/ HTTP/1.1
> [...]
> X-OCCI-Location:http://example.com/vms/user1/vm1, \
                  http://example.com/vms/user1/vm2, \
                  http://example.com/disks/user1/disk1

< HTTP/1.1 200 OK
< [...]
```

Removing a Resource Instance from a Mixin (Andy: See previous comment) One or multiple resource instances can be removed from a Mixin using the HTTP DELETE verb **(Andy: This 'feels' unRESTful - operations on either one or all resources at a URL)** . The URIs which uniquely defined the resource instance MUST be provided in the request:

```
> DELETE /my_stuff/ HTTP/1.1
> [...]
> X-OCCI-Location:http://example.com/vms/user1/vm1, \
                  http://example.com/vms/user1/vm2, \
                  http://example.com/disks/user1/disk1

< HTTP/1.1 200 OK
< [...]
```

3.2.2.4 Operation on Paths in the Namespace (Andy: Isn't the namespace just a URL pointing to resource(s)?

The following operations are defined when operating on paths in the namespace hierarchy which are not location paths nor resource instances **(Andy: unclear sentence - what's the intent?)**. They MUST end with / (For example *http://example.com/vms/user1*).

Retrieving All Resource Instances Below a Path The HTTP verb GET must be used to retrieve all resource instances. The service provider MUST return a URI-list **(Andy: noob comment 'what's that?!')** containing all resource instances which are children of the provided URI in the namespace hierarchy:

```
> GET /vms/user1/ HTTP/1.1
> [...]

< HTTP/1.1 200 OK
< Content-type: text/uri-list
< [...]
<
< http://example.com/vms/user1/vm1
< http://example.com/vms/user1/vm2
```

Note: A Service provider SHOULD support a filtering mechanism. If a category is provided in the request the server SHOULD only return the resource instances belonging to the provided Category.

Deletion all resource instances below a path **(Andy: this is a potentially dangerous operation!)** Ralf: +10 on that! The DELETE operation is very dangerous. DELETE used on a Mixin location only disassociates a resource instance from the Mixin. The same operation on a non-Mixin-location would remove everything there. This is especially dangerous with user-defined Mixins since the effect of the DELETE operation on different paths can change at run-time. The HTTP verb DELETE must be used to delete all resource instances under a hierarchy:

```
> DELETE /vms/user1/ HTTP/1.1
> [...]

< HTTP/1.1 200 OK
< [...]
```

Note: A Service provider SHOULD support a filtering mechanism. If a category is provided in the request the server SHOULD only return the resource instances belonging to the provided Category.

3.3 Syntax and Semantics of the Rendering

(Andy: Perhaps should be before previous section?) The following subsections demonstrate how the OCCI base types can be syntactically rendered.

Ralf: I would suggest talking about "HTTP Category Header" instead of "OCCI-Category", "HTTP Link Header" instead of "OCCI-Links", etc. Then each section can describe how the particular HTTP header is used by the OCCI HTTP rendering.

3.3.1 Rendering of an OCCI-Category

(Andy: Should we deal with/reference multiple header concatenation here? It's pretty important not to be hidden out of context.) The semantics of the Category in the OCCI context is described in the OCCI Core & Models document. This rendering follows the Category header defined by the Web Categories specification [5] and MUST be rendered accordingly.

```
Category: <term>; scheme="<scheme>"
    [;rel=<space-separated list of related Category identifiers>]
    [;attributes=<space-seperated list of attribute names>]
    [;title=<Title of this Category>]
    [;location=<Parent location>]
```

There is NO order for the optional part **(Andy: What 'optional' part?)**.

[5] http://tools.ietf.org/html/draft-johnston-http-category-header-01

3.3.2 Rendering of OCCI-Links and OCCI-Actions

The semantics of the Link header in the OCCI context is described in the OCCI Core & Models document. This rendering follows the Link header defined by the Web Linking specification [6] and MUST be rendered accordingly.

```
Link: <Resource URL>;
    rel=<space-separated list of Category identifiers of the target Resource type>
    [;self=<Link instance URL>]
    [;category=<space-separated list of Category identifiers of the Link type>
    [;<attribute name>=<attribute value>] ... ]

or in case it is an Action:

Link: <Resource URL> + ";action=" + <Term of the Action>;
    rel=<Category identifier of the Action>
```

3.3.3 Rendering of OCCI-Attributes

The X-OCCI-Attribute MUST be used to render the attributes associated with a OCCI Kind. A simple key-value format is used. The field value consist of an attribute name followed by an equal sign ("=") and the attribute value. The attribute value must be quoted **(Andy: double or single?)** if it includes a separator character, see RFC 2616 (page 16).

```
X-OCCI-Attribute: <attribute name>=<attribute value>
```

Valid attribute names for OCCI Kinds are specified in appropriate Extension documents **(Andy: ref infrastructure)**.

3.3.4 Rendering of Location-URLs

To render an OCCI representation solely in the header, the X-OCCI-Location HTTP header MUST be used to return a list of Kind URLs. Each header field value correspond to a single URL. Multiple Kind URLs are returned using multiple X-OCCI-Location headers. See RFC 2616 for information on how to render multiple HTTP headers.

```
X-OCCI-Location: <URL>
```

3.3.5 Fields

(Andy: a little unclear - maybe add 'the required HTTP headers are those needed at a minimum to represent the particular OCCI core model entity') Ralf: I like this table, having a clear overview of which HTTP Header is used to render a particular OCCI type/instance is very good in my opinion. Table need to be updated though. The following setups show how the Core Model MUST be rendered. Shown are the fields which MUST be available in a request from the Client or a response from the Server.

Operation	Required HTTP-Header(s)	Optional HTTP-Header(s)	Notes
Rendering of a Category instance	Category	N/A	
Rendering of a Kind instance	Category	N/A	
Rendering of a Mixin instance	Category	N/A	
Rendering a list of Category, Kind and Mixin instances	Category	N/A	
Rendering a list of Entity sub-type instances	X-OCCI-Location	N/A	
Rendering of a Resource	Category	X-OCCI-Attribute, Link	
Rendering of an Action	Category, Link	X-OCCI-Attribute	
Rendering of a Link	Category, Link	X-OCCI-Attribute	

[6] http://tools.ietf.org/html/draft-nottingham-http-link-header-10

3.4 General HTTP Behaviour Adopted by OCCI

The following sections deal with some general HTTP features which are adopted by OCCI.

(Andy: If this is general in respect to OCCI and in the context of HTTP then we should include the use of caching-related headers)

3.4.1 Security and Authentication

OCCI does not require that an authentication mechanism be used nor does it require that client to service communications are secured. It does recommend that an authentication mechanism be used and that where appropriate, communications are encrypted using HTTP over TLS. The authentication mechanisms that CAN be used with OCCI are those that can be used with HTTP and TLS, for example Basic **??**, Digest **??** and OAuth **??**. If an OCCI service requires authentication the response to a request that MUST be authenticated must be a HTTP 401 code that indicates the request is authorised. In response to authenticate the client MUST set a WWW-Authenticate header field and through this indicate the authentication mechanism.

3.4.2 Versioning

Information about what version of OCCI is supported by a provider MUST be advertised to a client on each response to a client. The version field in the response MUST include the value OCCI/X.Y, where X is the major version number and Y is the minor version number of the implemented OCCI specification. In the case of a HTTP Header Rendering, the server response MUST relay versioning information using the HTTP header name 'Server'.

```
HTTP/1.1 202 Accepted
Server: occi-server/1.1 (linux) OCCI/1.0
[...]
```

Complimenting the service-side behaviour of an OCCI implementation, a client MUST indicate to the OCCI service implementation the version it expects to interact with. For the clients, the information MUST be advertised in all requests it issues. In the case of a HTTP Header Rendering, the client request MUST relay versioning information in the 'User-Agent' header. The 'User-Agent' field MUST include the same value (OCCI/X.Y) as supported by the Server HTTP header.

```
(Andy: what's <UDN>?)

GET <UDN> HTTP/1.1
Host: example.com
User-Agent: occi-client/1.1 (linux) libcurl/7.19.4 OCCI/1.0
[...]
```

If a service receives a request from a client that supplies a version number higher than the service supports, the service MUST respond back to the client with an exception indicating that the requested version is not implemented. Where a client implements OCCI using a HTTP transport, the HTTP code 501, not implemented, MUST be used.

3.4.3 Content-type and Accept headers

Ralf: Add some HTTP examples for the different Content-types. Just use the same style as in the other sections.

A server MUST react according to the Accept header the client provides. If none is given - or **/** is used - the service MUST use the Content-type *text/plain*. This is the fall-back rendering and MUST be implemented.

Otherwise the according rendering MUST be used. Each Rendering SHOULD expose which Accept and Content-type header fields it can handle. Overall the service MUST support the *text/occi*, *text/plain* and *text/uri-list* Content-types.

The server MUST also return the proper Content-type header. If a client provides information with a Content-Type - the information MUST be parsed accordingly.

When the Client request a Content-Type that will result in an incomplete or faulty rendering the Service MUST return the unsupported media type , 415, HTTP code.

3.4.3.1 The Content-type text/plain While using this rendering with the Content-Type *text/plain* the information described in section 3.3 MUST be placed in the HTTP Body.

Each rendering of an OCCI base type will be placed in the body. Each entry consists of a name followed by a colon (":") and the field value. The format of the field value is specified separately for each of the three header fields, see section 3.3 **(Andy: is it not section 3.2?)**.

3.4.3.2 The Content-type text/occi While using this rendering with the Content-Type *text/occi* the information described in section **??** MUST be placed in the HTTP Header. The body MUST contain the string 'OK' on successful operations. **(Andy: and on failures?)**.

The HTTP header fields MUST follow the specification in RFC 2616 [**?**]. A header field consists of a name followed by a colon (":") and the field value. The format of the field value is specified separately for each of the header fields, see section 3.3.

Limitations: HTTP header fields MAY appear multiple times in a HTTP request or response. In order to be OCCI compliant the specification of multiple message-header fields according to RFC 2616 MUST be fully supported. In essence there are two valid representation of multiple HTTP header field values. A header field might either appear several times or as a single header field with a comma-separated list of field values. Due to implementation issues in many web frameworks and client libraries it is RECOMMENDED to use the comma-separated list format for best interoperability.

HTTP header field values which contain separator characters MUST be properly quoted according to RFC 2616.

Space in the HTTP header section of a HTTP request is a limited resource. By this, it is noted that many HTTP servers limit the number of bytes that can be placed in the HTTP Header area. Implementers MUST be aware of this limitation in their own implementation and take appropriate measures so that truncation of header data does NOT occur.

3.4.3.3 The Content-type text/uri-list This Rendering can handle the *text/uri-list* Accept Header. It will use the Content-type *text/uri-list*.

This rendering cannot render resource instances or Kinds or Mixins directly but just links to them. For concrete rendering of Kinds and Categories the Content-types *text/occi*, *text/plain* MUST be used. If a request is done with the *text/uri-list* in the Accept header, while not requesting for a Listing a Bad Request MUST be returned.

3.4.4 Return codes

At any point the service provider CAN return any of the following HTTP Return Codes:

Note: Other return codes can also be used if properly used. **(Andy: possibly not useful - might encourage behaviour not helpful to interop)**

Table 2. HTTP Return Codes

Code	Description	Notes
200	OK	
202	Accepted	For example when creating a Virtual machine the operation can take a while.
400	Bad Request	For example on parsing errors or missing information
401	Unauthorized	
403	Forbidden	
405	Method Not Allowed	
409	Conflict	
410	Gone	
415	Unsupported Media Type	
500	Internal Server Error	
501	Not Implemented	
503	Service Unavailable	

4 Contributors

Editors: Andy Edmonds, Thijs Metsch
Contributors: Alexander Papaspyrou, Ralf Nyrén, Sam Johnston

We would like to thank the following people who contributed to this document:

Name	Affiliation	Contact
Michael Behrens	R2AD	behrens.cloud at r2ad.com
Andy Edmonds	Intel - SLA@SOI project	andy at edmonds.be
Sam Johnston	Google	samj at samj.net
Gary Mazzaferro	OCCI Counselour - Exxia, Inc.	garymazzaferro at gmail.com
Thijs Metsch	Platform Computing, Sun Microsystems	tmetsch at platform.com
Ralf Nyrn	Aurenav	ralf at nayren.net
Alexander Papaspyrou	TU-Dortmund	alexander.papaspyour at tu-dortmund.de
Shlomo Swidler	Orchestratus	shlomo.swidler at orchestratus.com

Next to these individual contributions we value the contributions from the OCCI working group.

5 Glossary

Term	Description
Action	An OCCI base type. Represent an invocable operation on a Entity sub-type instance or collection thereof.
Category	A type in the OCCI model. The parent type of Kind.
Client	An OCCI client.
Collection	A set of Entity sub-type instances all associated to a particular Kind or Mixin instance.
Entity	An OCCI base type. The parent type of Resource and Link.
Kind	A type in the OCCI model. A core component of the OCCI classification system.
Link	An OCCI base type. A Link instance associate one Resource instance with another.
mixin	An instance of the Mixin type associated with a **resource instance**. The “mixin” concept as used by OCCI *only* applies to instances, never to Entity types.
Mixin	A type in the OCCI model. A core component of the OCCI classification system.
OCCI	Open Cloud Computing Interface
OCCI base type	One of Entity, Resource, Link or Action.
OGF	Open Grid Forum
Resource	An OCCI base type. The parent type for all domain-specific resource types.
resource instance	An instance of a sub-type of Entity. The OCCI model defines two sub-types of Entity, the Resource type and the Link type. However, the term *resource instance* is defined to include any instance of a *sub-type* of Resource or Link as well.
Tag	A Mixin instance with no attributes or actions defined.
Template	A Mixin instance which if associated at resource instantiation time pre-populate certain attributes.
type	One of the types defined by the OCCI model. The OCCI model types are Category, Kind, Mixin, Action, Entity, Resource and Link.
concrete type/sub-type	A concrete type/sub-type is a type that can be instantiated.
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
URN	Uniform Resource Name

6 Intellectual Property Statement

7 Disclaimer

8 Full Copyright Notice

9 References