GWD-R
OCCI-WG

Thijs Metsch, Platform Computing
Andy Edmonds, Intel
October 7, 2010
Updated: January 18, 2011

Open Cloud Computing Interface - RESTful HTTP Rendering

Status of this Document

This document provides information to the community regarding the specification of the Open Cloud Computing Interface. Distribution is unlimited.

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.

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 [1].

OCCI HTTP Rendering defines how to interact with the OCCI Core Model using the RESTful OCCI API [2]. 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 [3]. 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 [4].

This document uses the Augmented Backus-Naur Form (ABNF) notation of RFC 2616 [5], and explicitly includes the following rules from it: quoted-string, token, SP (space), LOALPHA, DIGIT.

All examples in this document use one of the following three HTTP category definitions. An example name-space hierarchy 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:

```
Category: compute;
          scheme="http://schemas.ogf.org/occi/infrastructure#";
```

[1] Infrastructure as a Service

```
          class="kind";
          location=/compute/
    (This is an compute kind)

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

Category: my_stuff;
          scheme="http://example.com/occi/my_stuff#";
          class="mixin";
          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/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/
```

The following terms [6] are used when referring to URI components:

```
 http://example.com:8080/over/there?action=stop#xyz
 \__/   \_____________/\_________/ \_________/ \_/
  |           |            |            |        |
scheme     authority       path        query   fragment
```

All examples in this document use the *text/plain* HTTP Content-Type for posting information. To retrieve information the HTTP Accept header *text/plain* is used.

This specification is aligned with RFC 3986 [6].

3 OCCI HTTP Rendering

The OCCI HTTP Rendering document specifies how the OCCI Core Model [1], including extensions thereof, is rendered over the HTTP protocol [5]. The document describes the general behavior for all interaction with an OCCI implementation over HTTP together with three content types to represent the data being transferred. The content types specified are *text/plain*, *text/occi* and *text/uri-list*. Other data formats such as e.g. OVF and JSON will be specified in complimentary documents.

3.1 A RESTful HTTP Rendering for OCCI

The OCCI HTTP Rendering uses all the features the HTTP and underlying protocols offer but also builds upon the Resource Oriented Architecture (ROA). ROA's use Representation State Transfer (REST) [7] 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).

Each resource instance within an OCCI system MUST be uniquely identified by an URI stored in the *id* attribute of the Entity type [1].

The structure of these URIs is opaque and the system should not assume a static, pre-determined scheme for their structure. For example *Entity::id* can be *http://example.com/vms/user1/vm1*.

3.2 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. OCCI service implementations MUST, at a minimum, support these verbs as shown in table 1.

Table 1. HTTP Verb Behavior

Type	GET	POST (create)	POST (action)	PUT (create)	PUT (update)	DELETE
resource instance	Rendering of this resource instance	Create a new resource instance	Trigger action	Create an resource instance at the given path	Update an resource instance at an given path	Delete this resource instance
Path in the name-space hierarchy ending with /	Listing of all resource instances below this name-space	Create a new resource instance	N/A	N/A	N/A	Delete all the resource instances below this name-space hierarchy
Location of an Mixin or Kind	Listing containing locations to all resource instances belonging to this Mixin or Kind	N/A	Trigger action (defined for this kind or mixin) on all resource instances belonging to this Mixin or Kind	Add an resource instance to a Mixin	N/A	Remove an resource instance given in the request from a Mixin
Query Interface	Listing of all registered Kinds and Mixins	N/A	N/A	Add a user defined Mixin	N/A	Remove a user-defined Mixin (defined in the request)

3.3 A RESTful Rendering of OCCI

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

3.3.1 Name-space Hierarchy and Location

The name-space and the hierarchy are free definable by the Service Provider. The OCCI implementation needs to implement the location path feature, which is required by OCCI for discovering capabilities and operations on Mixins and Kinds. Location paths tell the client where all resource instance of one Kind or Mixin (in case the Mixin is used as a tag) can be found regardless of the hierarchy the service provider defines. Location

paths are defined through a HTTP Category rendering and MUST be present for all HTTP Categories for which resource instance can be reated. The location paths MUST end with a '/'. These paths are discoverable by the client through the Query interface 3.4.1.

3.4 Various Operations and their Prerequisites and Behaviors

3.4.1 Handling the Query Interface

The query interface MUST be implemented by all OCCI implementations. It MUST be found at the path /-/ off the root of the OCCI implementation. The following operations, listed below, MUST be implemented by the service.

Retrieval of all registered Kinds and Mixins The HTTP verb GET must be used to retrieve all lKinds and Mixins the service can handle. This allows the client to discover the capabilities of the OCCI implementation. The result MUST contain all information about the Kinds and Mixins (including Attributes and Actions assigned).

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

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

An OCCI implementation MUST support a filtering mechanism. If a HTTP Category is provided in the request the server MUST only return the complete rendering of the requested Kind or Mixin.

Adding a Mixin definition To add a Mixin to the service the HTTP PUT verb MUST be used. All possible information for the Mixin must be defined. At least the HTTP Category term, scheme and location MUST be defined. Actions and Attributes are not supported:

```
> PUT /-/ HTTP/1.1
> [...]
> Category: my_stuff;
>           scheme="http://example.com/occi/my_stuff#";
>           class="mixin";
>           rel=http:/example.com/occi/something_else#mixin;
>           location=/my_stuff/;
```

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

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

Removing a Mixin definition A user defined Mixin MAY be removed (if allowed) 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#"; class="mixin";

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

3.4.2 Operation on Paths in the Name-space

The following operations are defined when operating on paths in the name-space hierarchy which are not location paths nor resource instances. 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 Listing containing all resource instances which are children of the provided URI in the name-space hierarchy:

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

< HTTP/1.1 200 OK
< [...]
<
< X-OCCI-Location: http://example.com/vms/user1/vm1
< X-OCCI-Location: http://example.com/vms/user1/vm2
```

An OCCI implementations MUST support a filtering mechanism. If a category is provided in the request the server MUST only return the resource instances belonging to the provided Mixin or Kind.

Deletion of all resource instances below a path (Note: this is a potentially dangerous operation!) 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
< [...]
```

3.4.3 Operations on Mixins or Kinds

All of the following operations MUST only be be performed on location paths provided by Kinds and Mixins. The path MUST end with an /.

Retrieving All Resource Instances Belonging to Mixin or Kind The HTTP verb GET must be used to retrieve all resource instances. The service provider MUST return a listing containing all resource instances which belong to the requested Mixin or Kind:

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

< HTTP/1.1 200 OK
< [...]
<
< X-OCCI-Location: http://example.com/vms/user1/vm1
< X-OCCI-Location: http://example.com/vms/user1/vm2
< X-OCCI-Location: http://example.com/vms/user2/vm1
```

An OCCI implementation MUST support a filtering mechanism. If a HTTP Category is provided in the request the server MUST only return the resource instances belonging to the provided Kind or Mixin. The provided HTTP category definition SHOULD be different from the Kind or Mixin definition which defined the location path used in the request.

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 Kind or Mixin which defines the location path which is used in the request:

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

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

Associate resource instances with Mixins 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
> X-OCCI-Location: http://example.com/vms/user1/vm2
> X-OCCI-Location: http://example.com/disks/user1/disk1

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

Unassociated resource instance(s) from a Mixin One or multiple resource instances can be removed from a Mixin using the HTTP DELETE verb. 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
> X-OCCI-Location: http://example.com/vms/user1/vm2
> X-OCCI-Location: http://example.com/disks/user1/disk1

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

3.4.4 Operations on Resource Instances

The following operations MUST be implemented by the OCCI implementation for operations on resource instances. The resource instance is uniquely identified by an URI (For example: *http://example.com/vms/user1/vm1*).[2]

[2]The path MUST not end with an '/' - that would mean that a client operates on a path in the name-space hierarchy

Creating a resource instance A request to create a resource instance MUST contain at least one HTTP category rendering which is (or relates to) a Kind definition. If multiple HTTP 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. Optional information which might be provided by the client and if available MUST be used are HTTP Links and HTTP X-OCCI-Attributes (mapping to Link and the attributes of an resource instance). Two ways can be used to create a new resource instance - HTTP POST or PUT:

```
> POST /compute/ HTTP/1.1
> [...]
>
> Category: compute; scheme="http://schemas.ogf.org/occi/infrastructure#"; class="kind";
> X-OCCI-Attribute: occi.compute.cores=2
> X-OCCI-Attribute: 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 MUST be any existing path in the hierarchy of the Service provider's name-space. It SHOULD be the location path of the corresponding Kind. The OCCI implementation MUST return the Location of the newly created resource instance.

HTTP PUT can also be used to create a resource instance. In this case the client ask the service provider to create a resource instance at a certain path in the name-space hierarchy.[3]

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

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

The OCCI implementation will return an OK code.

While creating a resource instance the resource instance is added to the collection defined by the Kind.

Retrieving a resource instance For retrieval the HTTP GET verb is used. It MUST return at least the HTTP category which defines the Kind of the resource instance. HTTP Links pointing to related resource instances, other URI or Actions MUST be included if present. Only Actions currently applicable SHOULD be rendered using HTTP Links. 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#"; class="kind";
< Category: my_stuff; scheme="http://example.com/occi/my_stuff#"; class="mixin";
< X-OCCI-Attribute: occi.compute.cores=2
< X-OCCI-Attribute: occi.compute.hostname=foobar
< Link: [...]
```

[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 3.6.6. Service Providers MUST ensure that the paths of REST resources stays unique in their name-space.

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 (HTTP Links, HTTP X-OCCI-Attributes or HTTP categories), which are updated MUST be provided along with the request.[4]

```
> 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.[5]

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

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

Triggering an Action on a resource instance To trigger an action on a resource instance the request MUST containing the HTTP Category defining the Action. It MAY include HTTP X-OCCI-Attributes which are the parameters of the action. Actions are triggered using the HTTP POST verb and by adding a query to the URI. This query exposes the term of the Action. If an action is not available a Bad Request should be returned.

```
> POST /vms/user1/vm1?action=stop HTTP/1.1
> [...]
> Category: compute; scheme="[...]"; class="action";
> X-OCCI-Attribute: method=poweroff

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

3.4.5 Handling Links resource instances

In general resource instance of the type Link and Resource are handled in the same way. Some special handling in the creation and handling of resource instance of the type Link are described in this section. They MUST be implemented by an OCCI implementation.

Creation of a Link during creation of a Resource instance When creating a resource instance of the type Resource, and Links are defined those Links MUST be created implicitly (Resulting in the creation of multiple REST resources. Still only the Locaction of the REST resource which represent the requested Kind MUST be returned - The URIs of the Links can be discovered by retrieving a rendering of the resource instance). To render all the Entity attributes of the Link those must be specified in the HTTP Link description during creation.

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

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

[5] If the resource instances is a Link type the source and target must be updated accordingly

```
> Category: compute;
            scheme="http://schemas.ogf.org/occi/infrastructure#";
            class="kind";
> Link: </network/123>;
        rel="http://schemas.ogf.org/occi/infrastructure#network";
        category="http://schemas.ogf.org/occi/infrastructure#networkinterface";
        occi.networkinterface.interface="eth0";
        occi.networkinterface.mac="00:11:22:33:44:55";
> X-OCCI-Attribute: occi.compute.cores=2
> X-OCCI-Attribute: occi.compute.hostname=foobar
> [...]

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

Retrieval resource instances of the type Resource with defined Links When an resource instance of the type Resource is rendered it MUST expose all the Links which are associated with the resource instance. Since Links are directed only those orgininating SHOULD be listed.

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

< HTTP/1.1 200 OK
< [...]
< Category: compute; scheme="http://schemas.ogf.org/occi/infrastructure#"; class="kind";
< Category: my_stuff; scheme="http://example.com/occi/my_stuff#"; class="mixin";
< X-OCCI-Attribute: occi.compute.cores=2
< X-OCCI-Attribute: occi.compute.hostname=foobar
< Link: </network/123>;
        rel="http://schemas.ogf.org/occi/infrastructure#network";
        self="/link/networkinterface/456";
        category="http://schemas.ogf.org/occi/infrastructure#networkinterface";
        occi.networkinterface.interface="eth0";
        occi.networkinterface.mac="00:11:22:33:44:55";
        occi.networkinterface.state="active";
```

Creation of Link resource instances To directly create a Link between two resource instance the Kind as well as a source and target attribute MUST be provide during creation of the resource instance (Which can be done with HTTP PUT or HTTP POST - See section 3.4.4 for a complete specification).

```
> POST /compute/ HTTP/1.1
> [...]
>
> Category: networkinterface;
            scheme="http://schemas.ogf.org/occi/infrastructure#";
            class="kind";
> X-OCCI-Attribute: source=http://example.com/vms/user1/vm1
> X-OCCI-Attribute: target=http://example.com/network/123
> [...]

< HTTP/1.1 200 OK
< [...]
< Location: http://example.com/link/networkinterface/456
```

Retrieval of Link resource instances Retrieval of a Link is analogue to the retrieval of any other resource instance. Please review section 3.4.4 for more details.

```
> GET /link/networkinterface/456 HTTP/1.1
> [...]

< HTTP/1.1 200 OK
< [...]
< Category: networkinterface; scheme="http://schemas.ogf.org/occi/infrastructure#"; class="kind
< X-OCCI-Attribute: occi.networkinterface.interface="eth0";
< X-OCCI-Attribute: occi.networkinterface.mac="00:11:22:33:44:55";
< X-OCCI-Attribute: occi.networkinterface.state="active";
< X-OCCI-Attribute: source=/vms/user1/vm1
< X-OCCI-Attribute: target=/network/123
```

3.5 Syntax and Semantics of the Rendering

All data transferred using the *text/occi* and *text/plain* content types is encoded with HTTP [5] compliant headers. Four specific HTTP headers are used:

- Category
- Link
- X-OCCI-Attribute
- X-OCCI-Location

The *text/occi* content type renders these headers as true HTTP headers in the header portion of a HTTP request or response. The *text/plain* content type renders the same headers, with identical syntax, in the body of the HTTP request/response. See section 3.6.5 for more information on the use of different content types.

Multiple HTTP header field values MUST be supported as defined by RFC 2616 [5]. This applies to both the *text/occi* and *text/plain* content types. RFC 2616 defines two different methods to render multiple header field values, either a comma-separated list or multiple header lines. The following two rendering examples are identical and both formats MUST be supported by both OCCI client and server to be compliant.

Comma-separated rendering of multiple HTTP header field values:

```
X-OCCI-Attribute: occi.compute.memory=2.0, occi.compute.speed=2.33
X-OCCI-Location: /compute/123, /compute/456
```

Separate header lines for each HTTP header field value:

```
X-OCCI-Attribute: occi.compute.memory=2.0
X-OCCI-Attribute: occi.compute.speed=2.33
X-OCCI-Location: /compute/123
X-OCCI-Location: /compute/456
```

3.5.1 Rendering of the OCCI Category, Kind and Mixin types

Instances of the Category, Kind and Mixin types [1] MUST be rendered using the Category header as defined by the Web Categories specification [6].

The following syntax applies:

```
Category           = "Category" ":" #category-value
  category-value   = term
                     ";" "scheme" "=" <"> scheme <">
```

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

```
                    ";" "class" "=" ( class | <"> class <"> )
                    *( ";" category-param )
  category-param   = ( ( "title" "=" quoted-string )
                   | ( "rel" "=" <"> resource-type <"> )
                   | ( "location" "=" URI )
                   | ( "attributes" "=" <"> attribute-list <"> )
                   | ( "actions" "=" <"> action-list <"> ) )
  term             = token
  scheme           = URI
  type-identifier  = scheme term
  resource-type    = type-identifier *( 1*SP type-identifier )
  class            = "action" | "mixin" | "kind"
  attribute-list   = attribute-name
                   | attribute-name *( 1*SP attribute-name)
  attribute-name   = attr-component *( "." attr-component )
  attr-component   = LOALPHA *( LOALPHA | DIGIT | "-" | "_" )
  action-list      = action
                   | action *( 1*SP action)
  action           = type-identifier
```

The following example illustrates a rendering of the Kind instance assigned to the Storage type [3]:

```
Category: storage;
    scheme="http://schemas.ogf.org/occi/infrastructure#";
    class="kind";
    title="Storage Resource";
    rel="http://schemas.ogf.org/occi/core#resource";
    location=/storage/;
    attributes="occi.storage.size occi.storage.state";
    actions="http://schemas.ogf.org/occi/infrastructure/storage/action#resize ...";
```

3.5.2 Rendering of OCCI Link instance references

The rendering of a Resource instance [1] MUST represent any associated Link instances using the HTTP Link header specified in the Web Linking RFC 5988 [8]. For example, rendering of a Compute instance linked to a Storage instance MUST include a Link header displaying the OCCI Link instance of the relation.

The following syntax MUST be used to represent OCCI Link type instance references:

```
Link               = "Link" ":" #link-value
  link-value       = "<" URI-Reference ">"
                     ";" "rel" "=" <"> resource-type <">
                     (";" "self" "=" <"> link-instance <">)
                     *( ";" link-param )
  link-param       = ( ( "category" "=" link-type )
                   | ( link-attribute ) )
  term             = token
  scheme           = URI
  type-identifier  = scheme term
  resource-type    = type-identifier *( 1*SP type-identifier )
  link-type        = type-identifier *( 1*SP type-identifier )
  link-instance    = URI-reference
  link-attribute   = attribute-name "=" ( token | quoted-string )
  attribute-name   = attr-component *( "." attr-component )
  attr-component   = LOALPHA *( LOALPHA | DIGIT | "-" | "_" )
```

The following example illustrates the rendering of a NetworkInterface [3] instance linking to a Network resource instance:

```
Link: </network/123>;
    rel="http://schemas.ogf.org/occi/infrastructure#network";
    self="/link/networkinterface/456";
    category="http://schemas.ogf.org/occi/infrastructure#networkinterface";
    occi.networkinterface.interface="eth0";
    occi.networkinterface.mac="00:11:22:33:44:55";
    occi.networkinterface.state="active";
```

3.5.3 Rendering of references to OCCI Action instances

The rendering of a Resource instance [1] MUST represent any associated Action instances using the HTTP Link header specified in the Web Linking RFC 5988 [8]. For example, rendering of a Compute instance MUST include a Link header displaying any Actions currently applicable to the resource instance.

The following syntax MUST be used to represent OCCI Action instance references:

```
Link               = "Link" ":" #link-value
  link-value       = "<" action-uri ">"
                      ";" "rel" "=" <"> action-type <">
  term             = token
  scheme           = URI
  type-identifier  = scheme term
  action-type      = type-identifier
  action-uri       = URL "?" "action=" term
```

The following example illustrates the rendering of a reference to the "start" Action defined for the Compute type [3]. Such a reference would be present in the rendering of a Compute instance.

```
Link: </compute/123?action=start>;
    rel="http://schemas.ogf.org/occi/infrastructure/compute/action#start"
```

3.5.4 Rendering of OCCI Entity attributes

Attributes defined for OCCI Entity sub-types [1], i.e. Resource and Link, MUST be rendered using the X-OCCI-Attribute HTTP header. For example the rendering of a Compute instance MUST render the associated attributes, such as e.g. `occi.compute.memory`, using X-OCCI-Attribute headers.

The X-OCCI-Attribute header uses a simple key-value format where each HTTP header field value represent a single attribute. The field value consist of an attribute name followed by the equal sign ("=") and an attribute value. The attribute value MUST be quoted if it contains a separator character as specified in RFC 2616 (page 16) [5].

The following syntax MUST be used to represent OCCI Entitiy attributes:

```
Attribute          = "X-OCCI-Attribute" ":" #attribute-repr
  attribute-repr   = attribute-name "=" ( token | quoted-string )
  attribute-name   = attr-component *( "." attr-component )
  attr-component   = LOALPHA *( LOALPHA | DIGIT | "-" | "_" )
```

Attribute names for the infrastructure types are defined in the OCCI Infrastructure document [3]. The rules for defining new attribute names can be found in the "Extensibility" section of the OCCI Core document [1].

The following example illustrates a rendering of the attributes defined by Compute type [3]:

```
X-OCCI-Attribute: occi.compute.architechture="x86_64"
X-OCCI-Attribute: occi.compute.cores=2
X-OCCI-Attribute: occi.compute.hostname="testserver"
X-OCCI-Attribute: occi.compute.speed=2.66
X-OCCI-Attribute: occi.compute.memory=3.0
X-OCCI-Attribute: occi.compute.state=active
```

3.5.5 Rendering of Location-URIs

In order to render an OCCI representation solely in the HTTP header, i.e. using the *text/occi* content type, the X-OCCI-Location HTTP header MUST be used to return a list of resource instance URIs. Each header field value correspond to a single URI. Multiple resource instance URIs are returned using multiple X-OCCI-Location headers.

```
Location      = "X-OCCI-Location" ":" location-value
  location-value  = URI-reference
```

The following example illustrates the rendering of a list of Compute resource instances:

```
X-OCCI-Location: http://example.com/compute/123
X-OCCI-Location: http://example.com/compute/456
X-OCCI-Location: http://example.com/compute/789
```

3.6 General HTTP Behaviors Adopted by OCCI

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

3.6.1 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 MAY be used with OCCI are those that can be used with HTTP and TLS.

3.6.2 Additional headers (Caching Headers)

The responses from an OCCI implementation MAY include additional headers like those for Caching purposes like E-Tags.

3.6.3 Asynchronous operations

OCCI implementations MAY implement a way to deal with asynchronous calls. Upon long-running operations the OCCI implementation might return an temporary result and the HTTP 202 return code. Client can pull upon this resource till the operation finishes. Upon completion of the operation this temporary result will redirect to the resulting REST resource using the HTTP 203 return code and a Location header.

3.6.4 Versioning

Information about what version of OCCI is supported by a OCCI implementation 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/1.1 200 OK
Server: occi-server/1.1 (linux) OCCI/1.1
[...]
```

Complimenting the service-side behavior of an OCCI implementation, a client SHOULD indicate to the OCCI service implementation the version it expects to interact with. For the clients, the information SHOULD be advertised in all requests it issues. A client request SHOULD 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.

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

If a OCCI implementation 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.

OCCI implementations which implement this version of the Document MUST use the version String *OCCI/1.1*.

3.6.5 Content-type and Accept headers

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* and *text/plain* 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.

The following examples demonstrate the behavior of an HTTP GET operations on the resource instance using two different HTTP Accept headers:

```
> GET /vms/user1/vm1 HTTP/1.1
> Accept: text/plain
> [...]

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

And with *text/occi* as HTTP Accept header:

```
> GET /vms/user1/vm1 HTTP/1.1
> Accept: text/occi
> [...]
```

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

3.6.5.1 The Content-type text/plain While using this rendering with the Content-Type *text/plain* the information described in section 3.5 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 3.5.

3.6.5.2 The Content-type text/occi While using this rendering with the Content-Type *text/occi* the information described in section 3.5 MUST be placed in the HTTP Header. The body MUST contain the string 'OK' on successful operations.

The HTTP header fields MUST follow the specification in RFC 2616 [5]. 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 3.5.

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.

3.6.5.3 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. Otherwise a list of resources must be rendered in *text/uri-list* format, which can be used for listing resource in collections or the namespace of the OCCI implementation.

3.6.6 Return codes

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

3.7 More complete examples

Since most examples are not complete due to space limitations this section will give some more complete examples.

Table 2. HTTP Return Codes

Code	Description	Notes
200	OK	
202	Accepted	Used for asynchronous non-blocking calls.
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	

3.7.1 Creating a compute resource instance

```
> POST / HTTP/1.1
> User-Agent: curl/7.21.0 (x86_64-pc-linux-gnu) libcurl/7.21.0 OpenSSL/0.9.8o zlib/1.2.3.4 libidn/1.
> Host: localhost:8080
> Accept: */*
> Cookie: pyocci_user=Zm9v|1291753962|7011a4821179ff98ea96d4b44fade0512b1ffc52
> Content-Type: text/occi
> Category: compute; scheme="http://schemas.ogf.org/occi/infrastructure#"; class="kind";
>
< HTTP/1.1 200 OK
< Content-Length: 2
< Content-Type: text/html; charset=UTF-8
< Location: /users/foo/compute/b9ff813e-fee5-4a9d-b839-673f39746096
< Server: pyocci OCCI/1.1
<
* Connection #0 to host localhost left intact
* Closing connection #0
OK%
```

3.7.2 Retrieving a compute resource instance

```
> GET /users/foo/compute/b9ff813e-fee5-4a9d-b839-673f39746096 HTTP/1.1
> User-Agent: curl/7.21.0 (x86_64-pc-linux-gnu) libcurl/7.21.0 OpenSSL/0.9.8o zlib/1.2.3.4 libidn/1.
> Host: localhost:8080
> Accept: */*
> Cookie: pyocci_user=Zm9v|1291753962|7011a4821179ff98ea96d4b44fade0512b1ffc52
>
< HTTP/1.1 200 OK
< Content-Length: 510
< Etag: "ef485dc7066745cb0fe1e31ecdd4895c356b5bd5"
< Content-Type: text/plain
< Server: pyocci OCCI/1.1
<
Category: compute;
    scheme="http://schemas.ogf.org/occi/infrastructure#"
    class="kind";
Link: </users/foo/compute/b9ff813e-fee5-4a9d-b839-673f39746096?action=start>;
    rel="http://schemas.ogf.org/occi/infrastructure/compute/action#start"
X-OCCI-Attribute: occi.compute.architecture=x86
X-OCCI-Attribute: occi.compute.state=inactive
```

```
X-OCCI-Attribute: occi.compute.speed=1.33
X-OCCI-Attribute: occi.compute.memory=2.0
X-OCCI-Attribute: occi.compute.cores=2
X-OCCI-Attribute: occi.compute.hostname=dummy
```

4 Contributors

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 Nyrén	Aurenav	ralf at nyren.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

The OGF takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the OGF Secretariat.

The OGF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to practice this recommendation. Please address the information to the OGF Executive Director.

7 Disclaimer

This document and the information contained herein is provided on an "As Is" basis and the OGF disclaims all warranties, express or implied, including but not limited to any warranty that the use of the information herein will not infringe any rights or any implied warranties of merchantability or fitness for a particular purpose.

8 Full Copyright Notice

References

[1] T. Metsch, A. Edmonds, and R. Nyrén, "Open Cloud Computing Interface – Core," http://ogf.org/gf/docs/, Nov. 2010, in Public Comment.

[2] T. Metsch and A. Edmonds, "Open Cloud Computing Interface – HTTP Rendering," http://ogf.org/gf/docs/, Nov. 2010, in Public Comment.

[3] ——, "Open Cloud Computing Interface – Infrastructure," http://ogf.org/gf/docs/, Nov. 2010, in Public Comment.

[4] S. Bradner, "Key words for use in RFCs to Indicate Requirement Levels," RFC 2119 (Best Current Practice), Internet Engineering Task Force, Mar. 1997. [Online]. Available: http://www.ietf.org/rfc/rfc2119.txt

[5] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1," RFC 2616 (Draft Standard), Internet Engineering Task Force, Jun. 1999, updated by RFCs 2817, 5785. [Online]. Available: http://www.ietf.org/rfc/rfc2616.txt

[6] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax," RFC 3986 (Standard), Internet Engineering Task Force, Jan. 2005. [Online]. Available: http://www.ietf.org/rfc/rfc3986.txt

[7] T. R. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[8] M. Nottingham, "Web Linking," RFC 5988 (Proposed Standard), Internet Engineering Task Force, Oct. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5988.txt