GFD-P-R.185
OCCI-WG

Thijs Metsch, Platform Computing
Andy Edmonds, Intel
June 21, 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 a 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 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 consists 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. They do not require changes to the HTTP Rendering specifications as of this version of the specification.

The current specification consists of three documents. This specification describes version 1.1 of OCCI. 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.

[1] Infrastructure as a Service

2.1 Specification Examples

All examples in this document use one of the following three HTTP category definitions. An example URL name-space hierarchy is also given. Syntax and Semantics are explained in the remaining sections of the document. These Category examples do not strive to be complete but to show the features OCCI has:

```
Category: compute;
          scheme="http://schemas.ogf.org/occi/infrastructure#";
          class="kind";
          location="http://example.com/compute/"
          [...]
    (This is a compute kind)

Category: networkinterface;
          scheme="http://schemas.ogf.org/occi/infrastructure#";
          class="kind";
          location="http://example.com/link/networkinterface/"
          [...]
    (This is a storage link)

Category: my_stuff;
          scheme="http://example.com/occi/my_stuff#";
          class="mixin";
          location="http://example.com/my_stuff/"
          [...]
    (This is a mixin of user1)
```

The following URL name-space hierarchy is used in the examples:

```
http://example.com/-/
http://example.com/vms/vm3
http://example.com/vms/foo/vm1
http://example.com/vms/bar/vm1
http://example.com/compute/
http://example.com/link/networkinterface/
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
```

2.2 Specification Examples' Content-type

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*.

More details are discussed in section 3.6.6. Other data formats such as e.g. OVF and JSON will be specified in complimentary documents.

3.1 Introduction

The OCCI HTTP Rendering uses many of the features the HTTP and underlying protocols offer and 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 have a unique identifier stored in the *occi.core.id* attribute of the Entity type [1]. It is RECOMMENDED to use a Uniform Resource Name (URN) as the identifier stored in *occi.core.id*.

The structure of these identifiers is opaque and the system should not assume a static, pre-determined scheme for their structure. For example *occi.core.id* could be *urn:uuid:de7335a7-07e0-4487-9cbd-ed51be7f2ce4*.

3.2 Behavior of the HTTP Verbs

As OCCI adopts a ROA, REST-based architecture and uses HTTP as the foundation protocol, resource instances are interacted with through the four main HTTP verbs. OCCI service implementations MUST, at a minimum, support these verbs as shown in the summary table 1.

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 Resource instance URL Name-space Hierarchy and Location

The URL name-space (in this document referred to as name-space) and the URL hierarchy (in this document referred to as hierarchy) are freely definable by the Service Provider. The OCCI implementation MUST implement the location path feature, which is required by the OCCI Query Interface. Location paths tell the client where all resource instances of one Kind or Mixin can be found regardless of the hierarchy the service provider defines. Location paths are defined through the HTTP Category rendering and MUST be present for all HTTP Categories that can be instantiated (i.e. provisioned). The location paths MUST end with a '/'. These paths are discoverable by the client through the Query interface 3.4.1.

Table 1. HTTP Verb Behavior Summary (* = Supports filtering mechanisms)

Path	GET	POST	POST (action - Query = ?action=...)	PUT	DELETE
resource instance (/vms/foo/vm1)	Retrieval of the resource instance's representation	Partial update of the resource instance	Perform an action	Creation/Update of the resource instance, supplying the full representation of the resource instance	Deletion of the resource instance
Kind collection (/compute/)	Retrieve a collection of resource instances*	Create a new resource instance of this Kind	Performs actions on a collection of resource instances	Not Defined	Removal of a single, a subset of or all the resource instances from the kind collection
Mixin collection (/my_stuff/)	Retrieve a collection of resource instances*	Adds a resource instance to this collection	Performs actions on a collection of resource instances	Update of the collection supplying the full representation of it. Includes removal and addition of resources.	Removal of a single, a subset of or all the resource instances from the Mixin collection
query interface (/-/)	Retrieve capabilities*	Add a user-defined Mixin	Not Defined	Not Defined	Removal of a user-defined Mixin

3.4 Various Operations and their Prerequisites and Behaviors

For the expected responses to all the behaviors specified in the sections below, please refer to the HTTP return codes table in section 3.6.8. This part of the OCCI specification introduces a filter mechanism. Filtering means that the OCCI implementation MUST only return resources which match the filtering patterns defined in the request. This allows clients to be more specific in their request and limit the amount of data transferred. Clients MAY currently only filter on Category identifiers and resource instance attributes. For Category the client MUST supply a valid Category rendering. For attributes the client MUST supply and attribute with a specific value.

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. Implementations MAY also adopt RFC 5785 [8] compliance to advertise this location (see Section 3.6.7 for more details). With the help of the query interface it is possible for the client to determine the capabilities of the OCCI implementation he refers to. The following Query Interface operations are listed below.

Retrieval of all Registered Kinds, Actions and Mixins The HTTP verb GET MUST be used to retrieve all Kinds, Actions and Mixins the service can manage. This allows the client to discover the capabilities of the OCCI implementation. The result MUST contain all information about the Kinds, Actions 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";
<           title="Compute Resource type";
<           rel="http://schemas.ogf.org/occi/core#resource";
<           attributes="occi.compute.cores occi.compute.state{immutable} ...";
<           actions="http://schemas.ogf.org/occi/infrastructure/compute/action#stop ...";
<           location="http://example.com/compute/"
< Category: storage;
<           scheme="http://schemas.ogf.org/occi/infrastructure#";
<           class="kind";
<           title="Storage Resource type";
<           rel="http://schemas.ogf.org/occi/core#resource";
<           attributes="occi.storage.size{required} occi.storage.state{immutable}";
<           actions="...";
<           location="http://example.com/storage/"
< Category: start;
<           scheme="http://schemas.ogf.org/occi/infrastructure/compute/action#";
<           class="action";
<           title="Start Compute Resource";
<           attributes="method"
< Category: stop;
<           scheme="http://schemas.ogf.org/occi/infrastructure/compute/action#";
<           class="action";
<           title="Stop Compute Resource";
<           attributes="method"
< Category: my_stuff;
<           scheme="http://example.com/occi/my_stuff#";
<           class="mixin";
<           location="http://example.com/my_stuff/"
```

An OCCI implementation MUST support a filtering mechanism. If one or multiple Categories are provided in the request the server MUST only return the complete rendering of the requested Kinds or Mixins. The *text/occi* rendering SHOULD be used to define the filters in a request.

The information which needs to be present in the request and response are defined as following and MUST be implemented by an OCCI implementation.

```
request     = filter
  filter    = *( Category CRLF )

response        = categories
  categories    = *( Category CRLF )
```

Adding a Mixin Definition To add a Mixin[2] to the service the HTTP POST 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:

```
> POST /-/ 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/"
```

[2] This can be used to 'tag' a set of resource instances.

```
< 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 specified location, scheme, term or rel value the service provider MAY reject this operation.

The information which needs to be present in the request and response are defined as following and MUST be implemented by an OCCI implementation.

```
request     = mixin
  mixin     = 1( Category CRLF )

response    = N/A
```

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
< [...]
```

The information which needs to be present in the request and response are defined as following and MUST be implemented by an OCCI implementation.

If a client attempts to remove a provider-defined Mixin category, the implementation MUST respond with a HTTP error code of 403, Forbidden.

When a user-defined Mixin is removed all associations with resource instances must be removed by the implementation.

```
request     = mixin
  mixin     = 1( Category CRLF )

response    = N/A
```

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/foo/*).

Retrieving the State of the Name-space Hierarchy The HTTP verb GET MUST be used to retrieve the current state of the name-space hierarchy. It MAY include URIs of resource instances, Paths in the name-space as well as URLs for Kind and Mixin collections.

It is RECOMMENDED to use the *text/uri-list* Accept HTTP header for this request.

```
> GET /vms/ HTTP/1.1
> Accept: text/uri-list
> [...]
```

```
< HTTP/1.1 200 OK
< Content-type: text/uri-list
< [...]
<
http://example.com/vms/vm3
http://example.com/vms/foo/
http://example.com/vms/bar/
```

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/foo/ HTTP/1.1
> [...]

< HTTP/1.1 200 OK
< [...]
<
< X-OCCI-Location: http://example.com/vms/foo/vm1
< X-OCCI-Location: http://example.com/vms/bar/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.

If an OCCI Entity attribute (X-OCCI-Attribute) is provided in the request the server MUST only return the resource instances which have a matching attribute value. The *text/occi* rendering SHOULD be used to define the filters in a request.

The information which needs to be present in the request and response are defined as following and MUST be implemented by an OCCI implementation.

```
request      = filter
  filter     = *( Category CRLF )
               *( Attribute CRLF )

response                    = resource_representations
  resource_representations  = *( Location CRLF )
```

Deletion of All resource instances Below a Path [3] The HTTP verb DELETE MUST be used to delete all resource instances under a hierarchy:

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

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

request      = N/A

response     = N/A
```

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 /.

In contrast to the last section it is valid for the operations defined here to return a 204 HTTP response code with no content. This is used when wanting to represent an empty collection of Mixins or Kinds.

[3] Note: this is a potentially dangerous operation!

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/foo/vm1
< X-OCCI-Location: http://example.com/vms/foo/vm2
< X-OCCI-Location: http://example.com/vms/bar/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.

If an OCCI Entity attribute (X-OCCI-Attribute) is provided in the request the server MUST only return the resource instances which have a matching attribute value. The *text/occi* rendering SHOULD be used to define the filters in a request.

The information which needs to be present in the request and response are defined as following and MUST be implemented by an OCCI implementation.

```
request      = filter
  filter     = *( Category CRLF )
               *( Attribute CRLF )

response                    = resource_representations
  resource_representations  = *( Location CRLF )
```

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 and the request MUST contain the Category defining the Action. Additionally 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: stop; scheme="[...]"; class="action";
> X-OCCI-Attribute: method="poweroff"

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

The information which needs to be present in the request and response are defined as following and MUST be implemented by an OCCI implementation.

```
request              = action_definition
  action_definition  = 1( Category CRLF )
                       *( Attribute CRLF )

response     = N/A
```

Associate resource instances With a Mixin One or multiple resource instances can be associated with a Mixin using the HTTP POST verb. The URIs which uniquely define the resource instance MUST be provided in the request:

```
> POST /my_stuff/ HTTP/1.1
> [...]
> X-OCCI-Location: http://example.com/vms/foo/vm1

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

The information which needs to be present in the request and response are defined as following and MUST be implemented by an OCCI implementation.

```
request      = 1*( resources )
  resources  = *( Location CRLF )

response     = N/A
```

Full Update of a Mixin Collection A collection consisting of Mixins can be updated using the HTTP PUT verb. All URIs which are part of the collection MUST be provided along with the request. The URIs which uniquely define the resource instances MUST be provided in the request:

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

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

The information which needs to be present in the request and response are defined as following and MUST be implemented by an OCCI implementation.

```
request      = 1*( resources )
  resources  = *( Location CRLF )

response     = N/A
```

Dissociate 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 define the resource instance MUST be provided in the request:

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

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

The information which needs to be present in the request and response are defined as following and MUST be implemented by an OCCI implementation.

```
request      = 1*( resources )
  resources  = *( Location CRLF )

response     = N/A
```

3.4.4 Operations on resource instances

The following operations MUST be implemented by the OCCI implementation for operations on resource instances, i.e. instances of either OCCI Resource, OCCI Link or sub-types thereof. All resource instances MUST be handled equally (if not stated otherwise) independent of whether they are instances of the OCCI Resource type or the OCCI Link type. A resource instance is uniquely identified by an URI, for example *http://example.com/vms/foo/vm1*. [4]

Creating a resource instance A request to create a resource instance MUST contain one and only one HTTP Category rendering which refers to a specific Kind instance. This Kind MUST define the type of the resource instance. A request MAY also contain one or more HTTP Category renderings which refer to different Mixin instances. Any such Mixin instances MUST be applicable (if allowed) to the resource instance. A client MAY be REQUIRED to provide additional information in the request based on whether the Kind indicates *required* attributes (see 3.5.1) or not.

A resource instance can be created using two ways - 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/1.1 201 OK
< [...]
< Location: http://example.com/vms/foo/vm1
```

The path on which this POST verb is executed MUST be the location path of the corresponding Kind. The OCCI implementation MUST return the URL of the newly created resource instance in the HTTP Location header[5].

The information which needs to be present in the request and response are defined as following and MUST be implemented by an OCCI implementation.

```
request                    = 1*( resource_representation )
  resource_representation  = 1*( Category CRLF )
                             *( Link CRLF )
                             *( Attribute CRLF )

response                   = 0*1( resource_representation )
  resource_representation  = 1*( Category CRLF )
                             *( Link CRLF )
                             *( Attribute CRLF )
```

HTTP PUT can also be used to create a resource instance. In this case the client asks the service provider to create a resource instance at a unique path in the name-space hierarchy.[6]

```
> PUT /vms/foo/my_first_virtual_machine HTTP/1.1
> [...]
>
> Category: compute; scheme="http://schemas.ogf.org/occi/infrastructure#"; "class=kind";
```

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

[5] The HTTP Location header [5] must not be confused with X-OCCI-Location.

[6] 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 3.6.8. Service Providers MUST ensure that the paths of REST resources stays unique in their name-space.

```
> X-OCCI-Attribute: occi.compute.cores=2
> X-OCCI-Attribute: occi.compute.hostname="foobar"
> [...]

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

The information which needs to be present in the request and response are defined as following and MUST be implemented by an OCCI implementation.

```
request                    = 1*( resource_representation )
  resource_representation  = 1*( Category CRLF )
                             *( Attribute CRLF )

response                   = 0*1( resource_representation )
  resource_representation  = 1*( Category CRLF )
                             *( Link CRLF )
                             *( Attribute CRLF )
```

The OCCI implementation MAY either return 201 or 200 HTTP return codes. If the OCCI implementation returns the 200 HTTP response code the full representation (as described in 'Retrieving a resource instance' in this section) MUST be returned. **Please note** that the HTTP Location header used when the service returns the 201 HTTP response code is defined in RFC2616 [5].

A created resource instance MUST be added to the collection defined by the Kind.

Retrieving a resource instance The HTTP GET verb is used for representation retrieval. It MUST return at least the HTTP Category which defines the Kind of the resource instance and associated attributes. HTTP Links pointing to related resource instances, other URI or Actions MUST be included if present. Only Actions currently applicable[7] SHOULD be rendered using HTTP Links. The Attributes of the resource instance MUST be exposed to the client if available.

```
> GET /vms/foo/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: [...]
```

The information which needs to be present in the request and response are defined as following and MUST be implemented by an OCCI implementation.

```
request          = N/A

response                  = resource_representation
resource_representation   = 1( Category CRLF )
                            *( Link CRLF )
                            *( Attribute CRLF )
```

[7] For example, it makes little sense to render the start action of a resource instance if it is already running.

Partial Update of a resource instance As this specification describes a RESTful service it is RECOMMENDED that the client first retrieves the resource instance. Partial updating is done using the HTTP POST verb. Only the information (HTTP Links, HTTP X-OCCI-Attributes or HTTP categories), which are updated MUST be provided along with the request. [8] If the resource instance updated is derived from the OCCI Link type HTTP Links MUST NOT be allowed in the request.

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

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

The information which needs to be present in the request and response are defined as following and MUST be implemented by an OCCI implementation.

```
request                    = 1*( resource_representation )
  resource_representation  = 1*( Category CRLF )
                             *( Link CRLF )
                             *( Attribute CRLF )

response     = N/A
```

The OCCI implementation MAY either return 201 or 200 HTTP return codes. If the OCCI implementation returns the 200 HTTP response code the full representation (as described in 'Retrieving a resource instance' in this section) MUST be returned. **Please note** that the HTTP Location header used when the service returns the 201 HTTP response code is defined in RFC2616 [5].

Full Update of a resource instance Before updating a resource instance it is RECOMMENDED that the client first retrieves the resource instance. Full updating is done using the HTTP PUT verb. The client must PUT the full representation, along with modifications, of the resource instance that the service supplied in the most recent GET. Missing information will result in the deletion of the same, if allowed by the implementation.

HTTP Links MUST NOT be allowed when using the HTTP PUT verb. A request containing a HTTP Link MUST result in a 400 Bad Request HTTP response. Any OCCI Links previously associated with a OCCI Resource MUST remain associated after a successful Full Update operation. This is necessary for the PUT operation to be idempotent.

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

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

The information which needs to be present in the request and response are defined as following and MUST be implemented by an OCCI implementation.

[8] Changing the type of the resource instance MUST NOT be possible.

```
request                    = 1*( resource_representation )
  resource_representation  = *( Category CRLF )
                             *( Attribute CRLF )

response     = N/A
```

The OCCI implementation MAY either return 201 or 200 HTTP return codes. If the OCCI implementation returns the 200 HTTP response code the full representation (as described in 'Retrieving a resource instance' in this section) MUST be returned. **Please note** that the HTTP Location header used when the service returns the 201 HTTP response code is defined in RFC2616 [5].

Deleting a resource instance A resource instance can be deleted using the HTTP DELETE verb. No other information SHOULD be added to the request.[9]

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

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

The information which needs to be present in the request and response are defined as following and MUST be implemented by an OCCI implementation.

```
request      = N/A

response     = N/A
```

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

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

The information which needs to be present in the request and response are defined as following and MUST be implemented by an OCCI implementation.

```
request              = action_definition
  action_definition  = 1( Category CRLF )
                       *( Attribute CRLF )

response     = N/A
```

[9] If the resource instances is an OCCI Link type the source and target Resources MUST be updated accordingly.

3.4.5 Handling Link instances

Some exceptions on the creation and handling of Link resource instances[10] are described in this section. They MUST be implemented by an OCCI implementation.

Inline Creation of a Link Instance When creating an instance of the OCCI Resource type and Links are defined in the request, those Links MUST be created implicitly. This results in the creation of multiple REST resources. However, only the Location of the REST resource which represents the requested Kind MUST be returned. The URIs of the Links can be discovered by retrieving a rendering of the resource instance. Attributes for the Link resource instance MUST be specified in the Link rendering during the creation of the resource instance. It is NOT recommended to supply the 'self' parameter to inline Link representation.

```
> POST /compute/ HTTP/1.1
> [...]
>
> 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/foo/vm1
```

Retrieving Resource instances With Defined Links When an resource instance of the OCCI Resource type is rendered it MUST expose all its owned Links. Since Links are directed only those originating outward SHOULD be listed.

```
> GET /vms/foo/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";
```

[10] A Link resource instance is an instance of the OCCI Link type or a sub-type thereof.

Creation of Link resource instances To directly create a Link between two existing resource instances the Kind as well as a occi.core.source and occi.core.target attribute MUST be provided during creation of the Link instance[11].

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

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

The information which needs to be present in the request and response are defined as following and MUST be implemented by an OCCI implementation.

```
request                    = resource_representation
resource_representation    = *( Category CRLF )
                             *( Attribute CRLF )

response                   = 0*1( resource_representation )
```

Retrieval of Link resource instances Retrieval of a Link is the same 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: occi.core.source="/vms/foo/vm1"
< X-OCCI-Attribute: occi.core.target="/network/123"
```

The information which needs to be present in the request and response are defined as following and MUST be implemented by an OCCI implementation.

```
request                    = N/A

response                   =  1*( resource_representation )
  resource_representation  = *( Category CRLF )
                             *( Attribute CRLF )
```

[11] See section 3.4.4

3.5 Syntax and Semantics of the Rendering

All data transferred using the *text/occi* and *text/plain* content types is structured text. These rendering structures are compliant with and follow the rules of HTTP headers [5]. Four specific rendering structures are only ever used:

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

The *text/occi* content type renders these rendering structures as HTTP headers in the header portion of a HTTP request or response. The *text/plain* content type renders the same rendering structures, with identical syntax, in the body of the HTTP request/response. See section 3.6.6 for more information on the use of different content types.

Multiple field values per rendering structure 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[12].

The following syntax applies:

```
Category             = "Category" ":" #category-value
  category-value     = term
                       ";" "scheme" "=" <"> scheme <">
                       ";" "class" "=" ( class | <"> class <"> )
                       [ ";" "title" "=" quoted-string ]
                       [ ";" "rel" "=" <"> type-identifier <"> ]
                       [ ";" "location" "=" <"> URI <"> ]
                       [ ";" "attributes" "=" <"> attribute-list <"> ]
                       [ ";" "actions" "=" <"> action-list <"> ]
  term               = LOALPHA *( LOALPHA | DIGIT | "-" | "_" )
  scheme             = URI
  type-identifier    = scheme term
  class              = "action" | "mixin" | "kind"
```

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

```
  attribute-list     = attribute-def
                     | attribute-def *( 1*SP attribute-def)
  attribute-def      = attribute-name
                     | attribute-name
                       "{" attribute-property *( 1*SP attribute-property ) "}"
  attribute-property = "immutable" | "required"
  attribute-name     = attr-component *( "." attr-component )
  attr-component     = LOALPHA *( LOALPHA | DIGIT | "-" | "_" )
  action-list        = action
                     | action *( 1*SP action )
  action             = type-identifier
```

The `attributes` parameter is used in the OCCI Query Interface, see section 3.4.1, to inform a client which attributes a particular Entity sub-type supports. This information is used by the client to determine which attributes to include in an initial POST request to create a Resource.

In addition to the existence of an attribute the Query Interface also informs a client whether an attribute is required or immutable, see the "attribute-property" in the syntax description above. The logic of attribute properties is as follows:

- If no attribute properties are defined the attribute is mutable and non-required, i.e. multiplicity is 0..x.
- If the "`immutable`" property is set the attribute is immutable (not modifiable by the client).
- If the "`required`" property is set the attribute has a multiplicity of 1..x, i.e. it MUST be specified by the client.

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{required} occi.storage.state{immutable}";
    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 [9]. 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 <"> ]
                      [ ";" "category" "=" link-type
                        *( ";" link-attribute ) ]
  term             = LOALPHA *( LOALPHA | DIGIT | "-" | "_" )
  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 [9]. 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             =  LOALPHA *( LOALPHA | DIGIT | "-" | "_" )
  scheme           = relativeURI
  type-identifier  = scheme term
  action-type      = type-identifier
  action-uri       = URI "?" "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 following syntax MUST be used to represent OCCI Entity attributes:

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

```
  number          = (int | float)
  int             = *DIGIT
  float           = *DIGIT "." *DIGIT
  bool            = ("true" | "false")
  enum_val        = string
```

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.architecture="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 [13] 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. For further discussion see Section 5.

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.

[13] This was introduced as the HTTP Location Header can have only one value where as this can have multiple.

3.6.3 Asynchronous Operations

OCCI implementations MAY implement a way to deal with asynchronous calls. Upon long-running operations the OCCI implementation MAY return a temporary resource (e.g. a task resource) using the HTTP Location header and a corresponding HTTP 202 return code. Clients can query that resource until the operation finishes. Upon completion of the operation this temporary result will redirect to the resulting REST resource using the HTTP Location header and return the HTTP 301 return code signalling the completion.

3.6.4 Batch operations

Batch operations, like the ones in described in section 3.4.3, are atomic. All parts of the request MUST be processed - no partial execution is allowed.

3.6.5 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. The server response MUST relay versioning information using the HTTP Server header.

```
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 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 HTTP Server header.

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

If an 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 this Document MUST use the version string *OCCI/1.1*. Versioning of extensions is out of scope for this version of the document.

3.6.6 Content-type and Accept headers

A server MUST react according to the Accept header the client provides. A client SHOULD provide the Accept header in a request. If none is given - or */* is used - the service MUST use the Content-type *text/plain*. This is the default and 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 *text/occi*, *text/plain* and *text/uri-list*.

The server MUST return the proper Content-type header. A client MUST provide the proper Content-Type when data is send to the OCCI implementation - the information MUST be parsed accordingly.

When the Client requests a Content-Type that will result in an incomplete or faulty rendering the Service MUST return the 406 'Not acceptable' 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/foo/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/foo/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
```

3.6.6.1 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.

3.6.6.2 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.

3.6.6.3 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 name-space of the OCCI implementation.

3.6.7 RFC5785 Compliance

Should implementations wish to advertise the Query Interface using the .well-known mechanism defined by "Defining Well-Known Uniform Resource Identifiers" [8] then they MUST use the following path served from the authority:

/.well-known/org/ogf/occi/-/

The functionality accessible at this location MUST exactly mirror that as defined in section 3.4.1 on the Query Interface.

3.6.8 Return Codes

At any point the service provider MAY return any of the following HTTP Return Codes in table. These codes are for behaviors in section 3.4 where appropriate 2

3.7 More Complete Examples

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

3.7.1 Creating a Compute resource instance

```
> POST /compute/ 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: */*
> 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/plain; charset=UTF-8
< Location: http://example.com/users/foo/compute/b9ff813e-fee5-4a9d-b839-673f39746096
< Server: example-occi OCCI/1.1
<
< 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: */*
>
< HTTP/1.1 200 OK
```

Table 2. HTTP Return Codes

Code	Description	Notes
200	OK	Indicates that the request was successful. The response MUST contain the created resource instance's representation.
201	OK	Indicates that the request was successful. The response MUST contain a HTTP Location header to the newly created resource instance.
202	Accepted	Used for asynchronous non-blocking calls. See section 3.6.3.
204	OK, but no content returned.	This is used to indicate that a collection is empty.
400	Bad Request	Used to signal parsing errors or missing information (e.g. an attribute that is required is not supplied in the request). This applies also to filters.
401	Unauthorized	The client does not have the required permissions/credentials.
403	Forbidden	Used to signal that a particular Mixin cannot be applied to a resource instance of a particular Kind. Used to signal that an attempt was made to modify an attribute that was marked as immutable.
404	Not Found	Used to signal that the request had information (e.g. a kind, mixin, action, attribute, location) that was unknown to the service and so not found.
405	Method Not Allowed	The service does not allow the client to issue the HTTP method against the requested path/location
406	Not Acceptable	See section 3.6.6
409	Conflict	A request contains content (e.g. mixin, kind, action) that results in an internal service, non-unique result (e.g. two types of start actions are found for Compute). The client MUST resolve the conflict by re-trying with specific Category information in the request.
410	Gone	A client attempts to retrieve a resource instance that no longer exists (i.e. it was deleted).
500	Internal Server Error	The state before the request should be maintained in such an error condition. The implementation MUST roll-back any partial changes made during the erroneous execution.
501	Not Implemented	If an implementation chooses not to implement a particular OCCI feature, it MUST signal the lack of that feature with this code. This implicitly points to a non-compliant OCCI implementation.
503	Service Unavailable	If the OCCI service is taken down for maintenance, this error code should be reported from the root of the name-space the provider uses.

```
< Content-Length: 642
< Etag: "ef485dc7066745cb0fe1e31ecdd4895c356b5bd5"
< Content-Type: text/plain
< Server: example-occi 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.core.id="urn:uuid:b9ff813e-fee5-4a9d-b839-673f39746096"
< X-OCCI-Attribute: occi.core.title="My Dummy VM"
< 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 OCCI Compliance Tools

For the ease of generating compliant parsers an ANTLR[14] grammar is available from[15] where it is updated and maintained. It is based on the ABNF grammars quoted through out the document. It can parse the OCCI text renderings of Category, Link, X-OCCI-Attribute and X-OCCI-Location. It is recommended that implementers use this grammar as a means to generate OCCI compliant parsers.

To verify an implementations run-time behavior, the OCCI Compliance Testing Tool is provided [16]. This tool uses the OCCI ANTLR generated parser along with various HTTP libraries to ensure that a OCCI implementation reacts and responds correctly to valid OCCI client requests. It is recommended that implementers use this tool to aid them in validating compliance with OCCI.

5 Security Considerations

The OCCI HTTP rendering assumes HTTP or HTTP-related mechanisms for security. As such, implementations SHOULD support TLS [17] for transport layer security.

Authentication SHOULD be realized by HTTP authentication mechanisms, namely HTTP Basic or Digest Auth [10], with the former as default. Additional profiles MAY specify other methods and should ensure that the selected authentication scheme can be renderable over the HTTP or HTTP-related protocols.

Authorization is not enforced on the protocol level, but SHOULD be performed by the implementation. For the authorization decision, the authentication information as provided by the mechanisms described above MUST be used.

Protection against potential Denial-of-Service scenarios are out of scope of this document; the OCCI HTTP Rendering specifications assumes cooperative clients that SHOULD use selection and filtering as provided by the Category mechanism wherever possible. Additional profiles to this document, however, MAY specifically address such scenarios; in that case, best practices from the HTTP ecosystem and appropriate mechanisms as part of the HTTP protocol specification SHOULD be preferred.

As long as specific extensions of the OCCI Core and Model specification do not impose additional security requirements than the OCCI Core and Model specification itself, the security considerations documented above apply to all (existing and future) extensions. Otherwise, an additional profile to this specification MUST be provided; this profile MUST express all additional security considerations using HTTP mechanisms.

[14] http://www.antlr.org
[15] http://github.com/dizz/occi-grammar
[16] http://forge.ogf.org/sf/scm/do/listRepositories/projects.occi-wg/scm
[17] http://datatracker.ietf.org/wg/tls/

6 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.

7 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
Mark Carlson	Oracle	mark.carlson at oracle.com
Andy Edmonds	Intel - SLA@SOI project	andy at edmonds.be
Sam Johnston	Google	samj at samj.net
Gary Mazzaferro	OCCI Counselour - AlloyCloud, 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 University	alexander.papaspyrou at tu-dortmund.de
Alexis Richardson	RabbitMQ	alexis at rabbitmq.com
Shlomo Swidler	Orchestratus	shlomo.swidler at orchestratus.com

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

8 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.

9 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.

10 Full Copyright Notice

References

[1] R. Nyrén, A. Edmonds, A. Papaspyrou, and T. Metsch, "Open Cloud Computing Interface – Core," GFD-P-R.183, April 2011. [Online]. Available: http://ogf.org/documents/GFD.183.pdf

[2] T. Metsch and A. Edmonds, "Open Cloud Computing Interface – HTTP Rendering," GFD-P-R.185, April 2011. [Online]. Available: http://ogf.org/documents/GFD.185.pdf

[3] ——, "Open Cloud Computing Interface – Infrastructure," GFD-P-R.184, April 2011. [Online]. Available: http://ogf.org/documents/GFD.184.pdf

[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, June 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, "Defining Well-Known Uniform Resource Identifiers (URIs)," RFC 5785 (Proposed Standard), Internet Engineering Task Force, Apr. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5785.txt

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

[10] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, E. Sink, and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication," RFC 2617 (Standard), Internet Engineering Task Force, 1999. [Online]. Available: http://www.ietf.org/rfc/rfc2617.txt