
OCCI Core Specification & Models

Introduction

The Open Cloud Computing Interface (OCCI) is an open protocol for cloud computing services. It initially targets the infrastructure services (IaaS) layer but its modular design which extends this OCCI Core Specification allows for future targeting of the platform services (PaaS) and application services (SaaS) layers on a future roadmap.

A Representational State Transfer (REST) protocol, it deviates from the underlying HyperText Transfer Protocol (HTTP) only where absolutely necessary and could be described as a "Resource Oriented Architecture (ROA)". Unlike other envelope-based protocols such as Atom and SOAP (which describe complex XML-based structures for data transferred within the HTTP payload, essentially duplicating HTTP's built in header-based metadata functionality), OCCI provides a "clean channel" over which any suitable format can travel without modification or wrapping, using HTTP to its full extent in the way it was intended. OCCI also uses various HTTP verbs in transactions rather than tunneling everything over POST, as was the case with SOAP and the various WS-* protocols. As such all existing HTTP features are available for caching, proxying, gatewaying and other advanced functionality such as partial GETs.

Each resource (i.e. a compute node) is identified by URL(s) and has one or more native representations (i.e. Open Virtualisation Format or OVF) as well as a generic XHTML5 rendering. The latter allows for direct end-user accessibility with well-formed, embedded semantic web markup for consumption by both human and machine clients. As such OCCI simultaneously presents both a machine interface (using native resource renderings) and a user interface (using HTML markup with forms and other web technologies such as Javascript/Ajax) so as to satisfy all common use cases. HTTP content negotiation is used to select between alternative representations and metadata including associations between resources is exposed via HTTP headers (e.g. the `Link:` and `Category:` headers).

In this way OCCI is not responsible for the representations themselves, rather it enables users to organise and group resources together to build arbitrarily complex systems of inter-related resources. It relies on existing standards for rendering and does not make any recommendations of one standard format over any other, in the same way as the Internet has many popular image formats.

Example

```
> GET /us-east/webapp/vm01 HTTP/1.1
> Host: cloud.example.com
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: application/ovf
< Link: </us-east/webapp/vm01;start>;
<       rel="http://purl.org/occi/action#start";
<       title="Start"
< Link: </us-east/webapp/build.pdf>;
<       rel="related";
<       title="Documentation";
<       type="application/pdf"
< Category: compute;
<       label="Compute Resource";
<       scheme="http://purl.org/occi/kind#"
< Server: occi-server/1.0 (linux) OCCI/1.0
< Connection: close
<
```

```
< <?xml version="1.0" encoding="UTF-8"?>
...
```

Essentials

Connection

Authentication

Servers *may* require that requests be authenticated using standard HTTP-based authentication mechanisms (including OAuth). OAuth They indicate this requirement by returning HTTP 401 with a WWW-Authenticate header and a suitable challenge (e.g. Basic, Digest, OAuth).

```
> GET / HTTP/1.1
> Host: cloud.example.com
>
< HTTP/1.1 401 Unauthorized
< WWW-Authenticate: OAuth realm="http://sp.example.com/"
```

The client then includes appropriate Authorization headers in its responses:RFC2617

```
> GET / HTTP/1.1
> Authorization: OAuth realm="http://sp.example.com/",
>                 oauth_consumer_key="0685bd9184jfhq22",
>                 oauth_token="ad180jjd733klru7",
>                 oauth_signature_method="HMAC-SHA1",
>                 oauth_signature="wOJIO9A2W5mFwDgiDvZbTSMK%2FPY%3D",
>                 oauth_timestamp="137131200",
>                 oauth_nonce="4572616e48616d6d65724c61686176",
>                 oauth_version="1.0"
```

Cookies

Servers *may* set and clients *may* accept *cookies* in order to maintain authentication state between requests:RFC2109

```
Set-Cookie: session=4732518c5fe6dbeb8429cdda11d65c3d; domain=.example.com; path=
```

Warning

Such sessions *should not* be used for other purposes (such as server-side state) in line with RESTful principles.

Versioning

Servers and clients *should* indicate the latest version of OCCI they support (e.g. 1.0) by way of the Server: and User-Agent: headers respectively, using the token "OCCI" (e.g. "OCCI/1.0"). If none is provided the latest (highest supported version number) available version offered by the server *should* be used:

```
> GET / HTTP/1.1
> Host: cloud.example.com
> User-Agent: occi-client/1.0 (linux) libcurl/7.19.4 OCCI/1.0
>
< HTTP/1.1 200 OK
```

```
< Server: Apache/2.2.13 (Unix) OCCI/1.0
```

Addressing

A single entry point is defined by a URL (e.g. `http://cloud.example.com/`) which may be a collection of resources or some other page as defined by the implementor (e.g. a marketing or landing page). All resources *must* be addressible by URLs (whose structure is opaque and at the discretion of the implementor) and discoverable via search and/or link traversal from the entry point. Clients *should not* rely on "rules" to construct URLs, rather learning them from URLs previously retrieved, in line with RESTful principles:

```
> GET /-/compute HTTP/1.1
> Accept: text/uri-list
>
< HTTP/1.1 200 OK
< Content-type: text/uri-list
<
< /node1
< /node2
```

Bootstrapping

Clients will typically conduct a GET or HEAD request on the root ("/") and discover the category search interface, from which they can learn the supported categories/kinds and retrieve some or all of them.

Example 1. Example bootstrap

Retrieve a collection of desired resources (having already discovered the category search URL and available categories):

```
# detect category search interface
> GET / HTTP/1.1
> Host: cloud.example.com
>
< HTTP/1.1 200 OK
< Content-type: text/html
< Link: </-/>; rel="search"; title="Category Search"
<
< <html>
< <head>
< <...>
<
# detect categories
> GET /-/ HTTP/1.1
>
< HTTP/1.1 200 OK
< Content-length: 0
<
< Category: compute; label="Compute Resource"; scheme="http://purl.org/occi/kin
< Category: network; label="Network Resource"; scheme="http://purl.org/occi/kin
< Category: storage; label="Storage Resource"; scheme="http://purl.org/occi/kin
< Category: us-east; label="US East Coast"; scheme="http://example.com/location
< Category: us-west; label="US West Coast"; scheme="http://example.com/location
< Category: demo; label="My Customer Demo"; scheme="http://example.com/~user/"
<
# retrieve category
> GET /-/compute HTTP/1.1
>
> HTTP/1.1 200 OK
> Content-type: text/uri-list
>
> /node1
> /node2
>
# retrieve resource
> GET /node1 HTTP/1.1
>
> HTTP/1.1 200 OK
> Content-type: application/ovf
> <...>
```

Tip

This discovery mechanism was selected so as to be compatible with existing content hosted at the same URL (e.g. <http://cloud.example.com>).

Operations

HTTP Verbs

Create, Retrieve, Update and Delete (CRUD) operations map to the POST, GET, PUT and DELETE HTTP verbs respectively. HEAD and OPTIONS verbs may be used to retrieve metadata and valid

operations without the entity body to improve performance. WebDAV definitions are used for MKCOL, MOVE and COPY.

Warning

Some providers may implement a subset of these operations, and those available to you for a given resource (if any) may depend on security policy. Be prepared to handle exceptions if you attempt to call operations that are not available to you.

POST (Create)	<p>“The POST method is used to request that the origin server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line.”RFC2616</p> <p>POSTing a representation (e.g. OVF) to a collection (e.g. /compute) will result in a new resource being created (e.g. /compute/123) and returned in the Location: header. POST is also used with HTML form data to trigger verbs (e.g. restart)</p>
GET (Retrieve - Metadata and Entity)	<p>“The GET method means retrieve whatever information (in the form of an entity) is identified by the Request-URI.”RFC2616</p> <p>GETting a resource (e.g. /compute/123) will return a representation of that resource in the most appropriate supported format specified by the client in the Accept header. Otherwise "406 Not Acceptable" will be returned.</p>
PUT (Create or Update)	<p>“The PUT method requests that the enclosed entity be stored under the supplied Request-URI.”RFC2616</p> <p>PUTting a representation (e.g. OVF) to a URL (e.g. /compute/123) will result in the resource being created or updated. The URL is known or selected by the client, in contrast to POSTs where the URL is selected by the server.</p>
DELETE (Delete)	<p>“The DELETE method requests that the origin server delete the resource identified by the Request-URI.”RFC2616</p> <p>DELETE results in the deletion of the resource (and everything "under" it, as appropriate).</p>

Tip

It is possible to instruct the server to create a resource based on a default configuration (without requiring client support) by doing an empty POST/PUT, specifying “Content-type: application/occi” (such that the web server knows where to route the request) and specifying the appropriate *kind* category (such that OCCI knows what to create).

Additionally the following HTTP methods are used:

COPY (Duplicate)	“The COPY method creates a duplicate of the source resource identified by the Request-URI, in the destination resource identified by the URI in the Destination header.”RFC4918
HEAD (Retrieve - Metadata Only)	“The HEAD method is identical to GET except that the server MUST NOT return a message-body in the response.”RFC2616
MKCOL (Make Collection)	“MKCOL creates a new collection resource at the location specified by the Request-URI.”RFC4918

MOVE (Relocate)	“The MOVE operation on a non-collection resource is the logical equivalent of a copy (COPY), followed by consistency maintenance processing, followed by a delete of the source, where all three actions are performed in a single operation.”RFC4918
OPTIONS	“The OPTIONS method represents a request for information about the communication options available on the request/response chain identified by the Request-URI.”RFC2616

Tip

Implementors may offer full WebDAV support in order to allow clients to enumerate the entire tree, interact with the resources via standard file managers (e.g. Windows Explorer, Mac OS X Finder), etc.

Actions

An *action* is some process that can be carried out on one or more *resources*, which may result in a state change and/or the creation of new resource(s).

Tip

Use common sense to decide what functionality should be exposed by way of actions and consult the list of existing actions and verbs before creating your own. For example it does not make sense to resize a storage resource by setting the “size” attribute (indeed there may not be space available or the filesystem may not support resizing and in any case the operation will take some time), nor to start a machine by changing the state from “stopped” to “running”.

Each available *action* for a given *resource* is indicated via a *link* with *class* extension set to “*action*” (such that clients can identify actions, including those from third-parties, without deriving meaning from the *rel* URI).

```
Link: </us-east/webapp/vm01;start>;
      rel="http://purl.org/occi/action#start";
      class="action";
      title="Start"
```

Actions defined by this standard reside under the `http://purl.org/occi/action#` namespace but anyone can define a new action by allocating a URI they control.

Warning

Defining your own actions can lead to interoperability problems and should be a last resort reserved for unique functionality. A simple peer review process is available for extending the registries which should be used where possible.

An *action* is triggered via an HTTP POST and depending on the action requested (e.g. *resize*), parameters *may* be provided using HTML forms (e.g. `application/x-www-form-urlencoded`). In the case of HTML-based renderings the actions can therefore be actual HTML forms.

Tip

Some resources can be interacted with but not rendered due to the nature of the resource or prevailing security policies (for example, an operator may be able to backup a machine without knowing anything about it).

Asynchronous Actions

Synchronous actions *may* return 200 OK on successful completion or 201 Created with a `Location:` header indicating a new resource for audit purposes.

Tip

Assume that clients are paranoid and want audit trails for all but the most trivial of actions.

In the event that the *action* does not complete immediately it *should* return HTTP 202 Accepted and a `Location:` header indicating a new resource where status and other pertinent information can be obtained.

Tip

Don't keep clients waiting - if you're not sure to return immediately then give them a resource they can monitor. For example by responding with an 202 Accepted return code and include a `location:` header, as described.

Advanced Actions

The specific parameters required and allowable values for them depend on the action and for advanced actions *may* require sending of custom *content types* rather than `application/x-www-form-encoded`.

State Machines

State machines are maintained on the server side and possible transitions are advertised to the client by way of action links. The links offered to a given client may depend on the resource, its current state, security policy, etc.

Tip

Many state transitions will not be effected immediately so be prepared to handle asynchronous responses.

Model

The model defines the objects and how they interrelate. An interface exposes "kinds" which have "attributes" and on which "actions" can be performed. The attributes are exposed as key-value pairs and applicable actions as links, following the REST hypertext constraint (whereby state transitions are defined *in-band* rather than via rules).

Kinds

Each category of resources distinguished by some common characteristic or quality is called a *kind* (e.g. `compute`, `network`, `storage`, `queue`, `application`, `contact`).

Kinds defined by this standard live in the `http://purl.org/occi/kind/` namespace but anyone can define a new kind by using a URI they control as the term.

Warning

Defining your own kinds can lead to interoperability problems and should be a last resort reserved for unique functionality. A simple peer review process is available for extending the registries which should be used where possible.

Each resource *must* specify a kind by way of a *category* within the *scheme* "`http://purl.org/occi/kind#`".

Tip

The word *type* is not used in this context in order to avoid confusion with Internet media types.

Attributes

An *attribute* is a specification that defines a property of an object. It is expressed in the form of key-value pairs. Attributes are divided into namespaces which are separated by the dot character (“.”).

Tip

This scalable approach was derived from the Mozilla Firefox `about:config` page.

Attributes defined by this standard reside at the root but anyone can define a new attribute by allocating a unique namespace based on their reversed Internet domain (e.g. “`com.example.attribute`”).

Warning

Defining your own attributes can lead to interoperability problems and should be a last resort reserved for unique functionality. A simple peer review process is available for extending the registries which should be used where possible.

Registry Entries

Table 1. Core Attributes

Attribute	Description	Type	Example
<code>id</code>	Immutable, unique identifier for the resource	URI	<code>urn:uuid:d0e9f0d0-f62d-4f28-bc90-23b0bd871770</code> or <code>urn:aws:ami-123456</code>
<code>title</code>	Display name for the resource	String	<code>Compute Resource #123</code>
<code>summary</code>	Description of the resource	String	<code>A virtual compute resource</code>

Categories

Category information allows for flexible organisation of resources into one or more vocabularies (each of which is referred to as a *scheme*).

The category model was derived from Atom and consists of three attributes:

<code>term</code>	The term itself (e.g. “ <code>compute</code> ”)
<code>scheme</code> (optional)	The vocabulary (e.g. “ <code>http://purl.org/occi/kind#</code> ”)
<code>label</code> (optional)	A human-friendly display name for the term (e.g. “ <code>Compute Resource</code> ”)

Category schemes and/or terms defined by this standard reside throughout the `http://purl.org/occi/` namespace but anyone can define a new scheme by allocating a URI they control.

Tip

Categories provide a flexible way to manage resources by taxonomy (categories) and/or folksonomy (tags), where both can be shared between [groups of] users or globally.

For example, users can create schemes for resource locations (e.g. US-East, US-West, Europe), operating systems (e.g. Windows, Linux) and patch levels (e.g.

Example 2. Category examples

OCCI kinds are represented by a category:

```
Category: compute;
        label="Compute Resource";
        scheme="http://purl.org/occi/kind#"
```

Implementors and users can also define their own vocabularies by defining schemes:

```
Category: cluster1;
        label="Cluster #1";
        scheme="http://example.com/clusters#"
```

Querying

The category query interface can be accessed by constructing an URL with the desired categories added to the path. Categories can be negated by prefixing with “-” and schemes may be specified with braces.

Example 3. Example category query

Locate the category search root (which *should* be `/-/`):

```
> HEAD / HTTP/1.1
>
< HTTP/1.1 200 OK
< Link: </-/>; rel="search"; title="Category Search"
```

Discover the available categories (which will all be returned in the same format as they appear in the HTTP headers):

```
> GET /-/ HTTP/1.1
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-type: application/occi
<
< Category: compute; label="Compute Resource"; scheme="http://purl.org/occi/kind#"
< Category: network; label="Network Resource"; scheme="http://purl.org/occi/kind#"
< Category: storage; label="Storage Resource"; scheme="http://purl.org/occi/kind#"
< Category: us-east; label="US East Coast"; scheme="http://example.com/location#"
< Category: us-west; label="US West Coast"; scheme="http://example.com/location#"
< Category: demo; label="My Customer Demo"; scheme="http://example.com/~user/"
```

Query the category search interface for the desired category(s):

```
> GET /-/compute/us-west HTTP/1.1
> Accept: text/uri-list
>
< HTTP/1.1 200 OK
< Content-type: text/uri-list
<
< /vm01
< /webapp/web01
< /webapp/web02
< /webapp/db01
```

Registry Entries

Table 2. Core Category Schemes

Scheme	Description	Example
<code>http://purl.org/occi/kind#</code>	OCCI Kinds	<code>compute</code>
<code>http://purl.org/occi/category#countries</code>	ISO Country Codes	<code>us</code>
<code>http://purl.org/occi/category#us-states</code>	US States	<code>ca</code>
<code>http://purl.org/occi/category#operating-systems</code>	Operating Systems	<code>linux</code>
<code>http://purl.org/occi/category#regulations</code>	Regulation compliance	<code>sox</code>

Other categories schemes can be added to the registry.

Collections

Where an operation could return multiple resources (e.g. categories, searches) this is referred to as a *collection*. Collections are returned as a list of URLs in `text/uri-list` format.RFC2483

Tip

Collections are passed by reference for simplicity rather than performance reasons, requiring $O(n+1)$ requests. Including metadata (via a wrapper format like Atom or SOAP) and/or the data itself would provide $O(1)$ performance, though this "pass by value" approach should only be considered where the representations are known to be small as encoding adds significant overhead.

Example 4. Example collection

```
# OCCI Example Collection
/examples/custom-extension
/examples/lamp-multi-vm
/examples/lamp
/examples/myservice
```

Paging

Collections *may* be divided into *pages*, with each linking to the “first”, “last”, “next” and “previous” *link relations*. The required *class* extension, with the value of *paging*, allows clients to group links together in the user interface and the server to specify e.g. "Next 10", "Next 100", etc.

```
Link: <http://example.com/xyz;start=0>; rel="first"; title="First"; class="paging"
Link: <http://example.com/xyz;start=400>; rel="previous"; title="Previous"; class="paging"
Link: <http://example.com/xyz;start=500>; rel="self"; title="Self"; class="paging"
Link: <http://example.com/xyz;start=600>; rel="next"; title="Next"; class="paging"
Link: <http://example.com/xyz;start=900>; rel="last"; title="Last"; class="paging"
```

Linking

Web linking standards for HTTP [LINK] and HTML [HTML5] are used to indicate associations between resources. All formats *must* support *in-band* linking including:

- Link relations (e.g. `rel="alternate"`)
- Pointers to resources (e.g. `href="http://example.com/"`)
- Internet media types (e.g. `type="text/html"`)
- Extensibility (e.g. `attribute="value"`)

```
Link: </us-east/webapp/build.pdf>;
      rel="related";
      title="Documentation";
      type="application/pdf"
```

Link relations defined by this standard reside under the `http://purl.org/occi/rel` namespace but anyone can define a new *link relation* by allocating a URI they control.

Registry Entries

Table 3. Core Link Relations

Relation	Description
first	“An IRI that refers to the furthest preceding resource in a series of resources.” [LINK]
help	“The referenced document provides further help information for the page as a whole.” [HTML5]
icon	“The specified resource is an icon representing the page or site, and should be used by the user agent when representing the page in the user interface.” [HTML5]
last	“An IRI that refers to the furthest following resource in a series of resources.” [LINK]
next	“A URI that refers to the immediately following document in a series of documents.” [LINK]
previous	“A URI that refers to the immediately preceding document in a series of documents.” [LINK]
search	“The referenced document provides an interface specifically for searching the document and its related resources.” [HTML5, OpenSearch]
self	“Identifies a resource equivalent to the containing element” [RFC4287]

Extensibility

The interface is fully extensible, both via a public peer review process (in order to update the specification itself, usually via registries) and via independent allocation of unique namespaces (in order to cater for vendor-specific enhancements).

Foreign markup

Implementations *must* accept and forward but otherwise ignore markup they do not understand.

Security Considerations

Encryption is not required by the specification in order to cater for sites that do not or can not use it (e.g. due to export restrictions, performance reasons, etc.), however SSL/TLS *should* be used over public networks including the Internet.

Glossary

in-band	“Sending of metadata and control information in the same band, on the same channel, as used for data”, for example, by embedding it in HTML. [http://en.wikipedia.org/wiki/In-band]
kind	“A category of things distinguished by some common characteristic or quality”, for example events, messages, media. [http://wordnetweb.princeton.edu/perl/webwn?s=kind]
out-of-band	“Communications which occur outside of a previously established communications method or channel”, for example, in HTTP headers. [http://en.wikipedia.org/wiki/Out-of-band_signaling]
type	Internet media (MIME) type.

Bibliography

Normative References

- [RFC2109] *RFC 2109 - HTTP State Management Mechanism*. <http://tools.ietf.org/html/rfc2109>. Internet Engineering Task Force (IETF) 1997-02.
- [RFC2483] *RFC 2483 - URI Resolution Services Necessary for URN Resolution*. <http://tools.ietf.org/html/rfc2483#section-5> [<http://tools.ietf.org/html/rfc2109>]. Internet Engineering Task Force (IETF) 1999-01.
- [RFC2616] *RFC 2616 - Hypertext Transfer Protocol -- HTTP/1.1*. <http://tools.ietf.org/html/rfc2616>. Internet Engineering Task Force (IETF) 1999-06.
- [RFC2617] *RFC 2617 - HTTP Authentication: Basic and Digest Access Authentication*. <http://tools.ietf.org/html/rfc2617> [<http://tools.ietf.org/html/rfc2616>]. Internet Engineering Task Force (IETF) 1999-06.
- [RFC3339] *RFC 3339 - Date and Time on the Internet: Timestamps*. <http://tools.ietf.org/html/rfc3339>. Internet Engineering Task Force (IETF) 2002-07.
- [RFC4918] *RFC 4918 - HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV)*. <http://tools.ietf.org/html/rfc4918> [<http://tools.ietf.org/html/rfc2616>]. Internet Engineering Task Force (IETF) 2007-06.
- [OpenSearch] *OpenSearch 1.1*. <http://www.opensearch.org/Specifications/OpenSearch/1.1> [<http://tools.ietf.org/html/rfc2616>]. A9.com, Inc. (an Amazon company) Clinton DeWitt. Joel Tesler. Michael Fagan. Joe Gregorio. Aaron Sauve. James Snell. 2009.

Informative References

- [RFC4287] *RFC 4287 - The Atom Syndication Format*. <http://tools.ietf.org/html/rfc4287>. Robert Syre. Mark Nottingham. Internet Engineering Task Force (IETF) 2005-12.
- [HTML5] *HTML 5*. <http://www.w3.org/TR/html5/> [<http://tools.ietf.org/html/rfc4287>]. Ian Hickson. David Hyatt. World Wide Web Consortium (W3C) 2009-08-25.
- [OAuth] *OAuth*. <http://oauth.net/core/1.0> [<http://tools.ietf.org/html/rfc2616>]. OAuth Core Workgroup <spec@oauth.net>. 2007-12-04.

[RWS] *RESTful Web Services*. <http://oreilly.com/catalog/9780596529260/>. 9780596529260. O'Reilly Media Leonard Richardson. Sam Ruby. 2007-05.

[LINK] *Web Linking*. <http://tools.ietf.org/html/draft-nottingham-http-link-header>. Internet Engineering Task Force (IETF) Mark Nottingham. 2009-07-12.

[CATEGORY] *Web Categories*. <http://tools.ietf.org/html/draft-johnston-http-category-header>. Internet Engineering Task Force (IETF) Sam Johnston. 2009-07-1.