Multi-Tenant Tenanting in OpenStack

Jorge L Williams <jorge.williams@rackspace.com>
Ziad N Sawalha <ziad.sawalha@rackspace.com>
Khaled Hussein <khaled.hussein@rackspace.com>

Abstract

As a cloud computing platform, OpenStack must support the concept of multi-tenancy. A common approach to organizing resources by 'tenant' across services is needed to be able to correlate usage tracking, auditing, authorization, and so forth. Within each multi-tenant service, the ability to identify each tenant's resources is also key.

The exact definition of a tenant and what it maps to in an operator's business model is unpredictable. Some operators will map tenants to customes, others to tenants (whatever tenant means for them), and others yet may map them to a cost center, and environment (production, staging, test, dev), etc... This document explains the rationale behind the lightweight standard for service developers adopted by OpenStack to implement tenancy and resource grouping without a-priori knowledge of billing, accountinging, and customer models and processes specific to the operator of an OpenStack deployment.

Table of Contents

Rationale and Goals

Building multi-tenant services is complicated and often involves knowledge of business processes that vary from one organization to another. We propose a method of organizing resources that allows multi-tenancy to be implemented on top of OpenStack services. By doing so we introduce a separation of concern between operators and service developers. Service developers offer management to tenants. From their perspective tenants are simply collection of resources. Operators manage tenants that may be associated with one or more accounts, customers, departments, or whateve their business model looks like. This approach lowers barriers to service developers by allowing them to develop services without a-priori knowledge of billing and accounting processes of the organization in which the services are deployed. Likewise, organizations will be given flexibility in the manner in which they deploy and offer OpenStack services. In this blueprint, we define a simple tenant admin API that facilitates and standardizes on this approach.

Specification Overview

Tenant Lifecycle

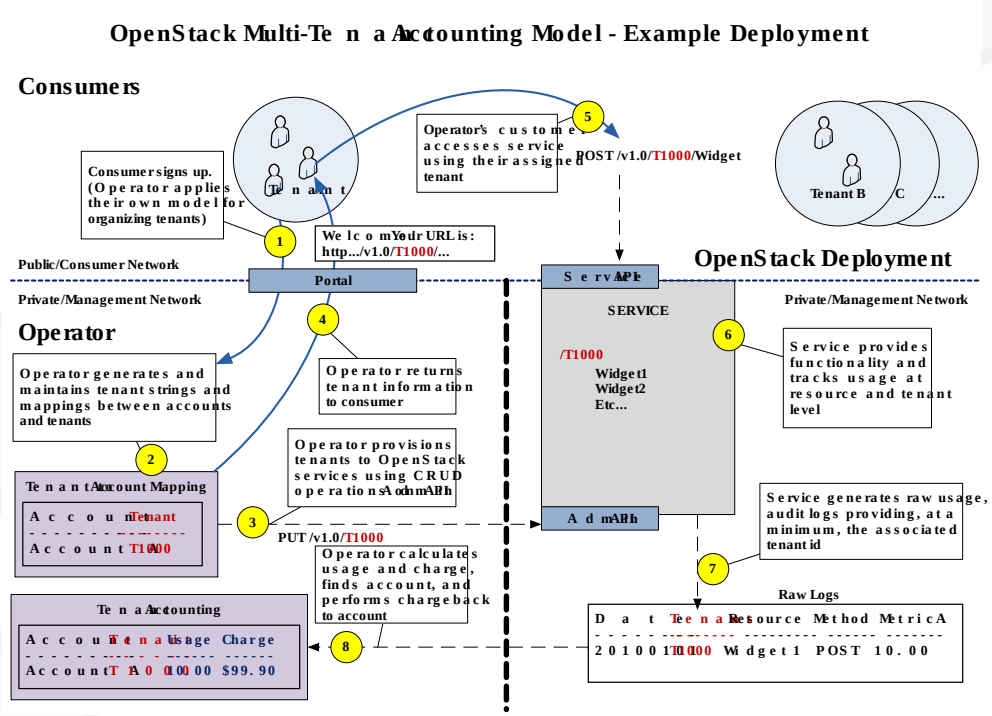
From the perspective of a service developer a tenant ID is simply an arbitrary string that is used to organize resources. We propose that a string be used as a top level resource collection after the version identifier: /

version/*tenantId*. Placing the tenant ID as a top level container dictates that all client requests are automatically associated with a tenant. Requests to create tenants or move resources between tenants are received via an admin API which is described in detail in the next section. Developers are responsible for ensuring that all usage metrics contain the tenant ID string.

Service operators, on the other hand, are responsible for organizing resources around tenants for the purposes of billing and authorization. Operators use tenant IDs to help organize service resources. The then expose service endpoints to their users and a method of tracking the tenant ID (Example, they may provide their tenants with API endpoints that contain the tenant ID embedded in the URI or, as an alternative, may track the tenant ID through the use of an authentication mechanism like tokens from OpenStack's Identity Service, called Keystone). The operator can then collect usage logs from the service and aggregate necessary usage metrics in order to charge back usage for the tenant to the appropriate entity (customer, account, department, cost center, etc...).

The relationships among tenants, operators, and services are illustrated in detail in the figure below.

Figure 1. Multi-Tenancy Overview



Admin API

A *service API* is an API that's made available to most clients — in most cases it is the public API that users consume. In contrast, an *admin API* is an implementation of the service API with additional calls to allow for the management and maintenance of the service. The admin API is consumed strictly by operators. Calls whose effects span multiple tenants should be placed in an admin API. Admin APIs **SHOULD NOT** be exposed via public endpoints and **SHOULD** have tighter security constraints than those of service APIs. We recommend that admin API users and service API users authenticate against separate authentication systems. All OpenStack services **MUST** implement an admin API.

Note

The requirement for an additional admin API does not necessarily dictate that two separate implementations of the API be written. Service teams may opt to write a single implementation of the API and expose it via two separate endpoints: a public endpoint and an admin endpoint. Alternatively, they may write one endpoint that exposes the administrative API calls to appropriately authorized clients. In the public endpoint, reverse proxy filters may be employed to cull admin calls before they reach the service implementation. A different authentication component may also be used at each endpoint to interact with separate authentication systems.

In the following sections, we propose a set of calls that **MUST** be implemented by admin APIs in OpenStack and an optional set that **SHOULD** be implemented. Together these calls allow for a simple and consistent admin API for the management of tenants in OpenStack.

Required Operations

The following operations **MUST** be implemented by OpenStack services and **MUST** be made available via the admin API. At their discretion, service operators **MAY** provide public access to **GET** and **HEAD** operations via the service API. The **PUT** and **DELETE** calls, however, **SHOULD** be accessible from the admin API only.

Get Tenant

Verb	URI	Description
GET	/*version*/*tenantId*	Get Tenant.

Normal Response Code(s): 200, 203, 204

Error Response Code(s): 404, 410, others …

Services are not required to provide a representation of a tenant on a **GET** request. If a representation is returned, it **SHOULD** provide information about the tenant along with tenant metadata. Additionally, the representation **MAY** contain a list of top level tenant resources. The actual format of the representation is service-specific.

If a service returns a tenant representation, it should return either a response code of 200 (Okay) or 203 (Non-Authoritative Information) if the request is cached. If a service does not return a representation, then it **MUST** return a 204 (No Content). Generally, a response code in the 200s signifies that the tenant exists and is valid. A 404 (Not Found) signifies that the tenant does not exist and a 410 (Gone) means that the tenant has recently been marked for deletion, is currently unavailable, and may be recoverable. Services may provide an additional operation to recover a recently removed tenant.

Get Tenant Metadata

Verb	URI	Description
HEAD	/*version*/*tenantId*	Get Tenant Metadata.

Normal Response Code(s): 204

Error Response Code(s): 404, 410, others …

A **HEAD** operation **MAY** return metadata for a tenant. If it does, it **MUST** return the same metadata that would be returned via a **GET** operation. The response to this call **MUST** only contain HTTP headers. As with **GET** requests, a 204 (No Content) signifies that the tenant exists and is valid. A 404 (Not Found) signifies that the tenant does not exist and a 410 (Gone) means that the tenant has recently been marked for deletion, is currently unavailable, and may be recoverable. Again, services may provide an additional operation to recover a recently removed tenant. The **HEAD** operation may be used as a shorthand for **GET** in cases where the service returns a representation document but the client is not interested in it.

Create a tenant

Verb	URI	Description
PUT	/*version*/*tenantId*	Create or Modify a tenant.

Normal Response Code(s): 201, 202,

Error Response Code(s): 409, others …

A **PUT** operation can be used to create or (optionally) modify a tenant. If a service provides a representation for a tenant, the representation **SHOULD** be included as part of the **PUT** request and it **SHOULD** match the representation returned by **GET**. One possible use of a tenant representation is to keep track of a tenant's tier in cases where the service offers different levels of performance at different tiers. Here, an operator may create a new tenant and assign it to a tier with a single **PUT** request. The operator may also update a tenant's tier by performing additional **PUT**s on the tenant. On success, a 201 (Created) should be returned when the tenant is created and a 202 (Accepted) should be returned when the tenant is modified.

In cases where the tenant representation offers a list of tenant resources, the **PUT** operation **SHOULD NOT** be used to add resources to or remove resources from the tenant. Services **MUST** ensure that **PUT** requests are idempotent. If a tenant does not have a representation, or the representation is not updatable, a 409 (Conflict) may be returned to indicate that a tenant with the given ID has already been created and may not be updated.

Note that a **PUT** operation is used to create a new tenant *with a tenant ID*. This means that the operator is in complete control of the tenant ID value and that the tenant ID is *not* generated by the service. That said, the following are the properties of an tenant ID that service implementers can rely on.

1. The tenant ID is a string in the UTF-8 character set.

2. The UTF-8 string will not be greater than 255 character units and it will not be empty.

3. The string may contain any character other than the path separator: `/`.

4. The UTF-8 string will be properly encoded in the request URL according to the encoding rules defined in RFC 1738 [1]. Services **MAY** reject improperly encoded URLs.

An OpenStack service should make no assumptions about the tenant ID other than those listed above. As a result, services **MUST** set aside 255 character units for storing tenant IDs. Services should also consider long tenant IDs when imposing limits on the size of a request URL.

The following are examples of tenant IDs and their encoded URLs:

Example 1. Example tenant IDs

tenantId	Sample Encoded URL	Valid
12345	https://widgets.openstack.com/v1.0/12345/widgets	Yes
Bob's Tenant	https://widgets.openstack.com/v1.0/Bob's%20Tenant/widgets	Yes
∑∞ΔΠ	https://widgets.openstack.com/v1.0/%E2%88%91%E2%88%9E%E2%88%86%E2%88%8F/widgets	Yes
resel:sub:acct	https://widgets.openstack.com/v1.0/resel1:sub2:acct3/widgets	Yes
resel\sub\acct	https://widgets.openstack.com/v1.0/resel1\sub2\acct3/widgets	Yes
resel/sub/acct	https://widgets.openstack.com/v1.0/resel1/sub2/acct3/widgets	No, using path separator.
	https://widgets.openstack.com/v1.0//widgets	No, empty.

The restrictions placed on tenant IDs **SHOULD** be described in the admin API documentation and **MAY** also be documented in the admin WADL. An example WADL is illustrated below.

Example 2. Tenant ID Sample WADL Definition

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

<application xmlns="http://wadl.dev.java.net/2009/02"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:w="http://widget.openstack.com/widget/api/v1.0">

   <grammars>
    <schema
        elementFormDefault="qualified"
        attributeFormDefault="unqualified"
        targetNamespace="http://widget.openstack.com/widget/api/v1.0"
        xmlns="http://www.w3.org/2001/XMLSchema">

        <simpleType name="TenantID">
            <restriction base="xsd:string">
                <pattern value="[^/]+" />❶
            </restriction>
        </simpleType>
    </schema>
   </grammars>

   <resources base="https://widget.openstack.com/widget/api/v1.0">
    <resource path="{tenantId}">
        <param name="tenantId" style="template" type="w:TenantID"/>  ❷
        .
        .
        .
    </resource>
   </resources>
</application>
```

❶ Note that the tenant ID pattern is very simple. Tenant IDs must contain one or more characters not matching the path separator: `/`.

❷ Here we define *`tenantId`* as a URI template parameter of type TenantID. The fact that we define the TenantID type so that it restricts the use of the path separator character is redundant in this case because template parameters do not allow values with path separators. Nonetheless, we define the TenantID type in order to be explicit and in case the type is used elsewhere.

Remove a tenant

Verb	URI	Description
DELETE	/*`version`*/*`tenantId`*	Remove a tenant.

Normal Response Code(s): 204

Error Response Code(s): 404, 410, others …

A **DELETE** operation is used to remove a tenant. a tenant's resources **SHOULD** be deleted after a tenant has been removed. That said, resources **SHOULD** remain recoverable and in a deleted state for a period of time before they are actually removed. This prevents data loss in cases involving human error. The **DELETE** operation **SHOULD** always return asynchronously. On success it should return a 204 (No Content). The operation should return a 404 (Not Found) if the tenant does not exist and a 410 (Gone) if the tenant has already been marked for deletion and is still in a recoverable state. Services may provide an additional operation to recover tenants that have been marked for deletion but have not yet been removed.

Optional Operations

The following operations **SHOULD** be implemented by OpenStack services, but it is not a strict requirement that services support them. The operations involve moving resources from one tenant to another. There are a number of use cases where such moves are necessary, and the operations below allow these use cases to be implemented in an efficient manner. If a service team should decide not to include support for the following calls it is recommended that, at the very least, a manual operational process exists that provides the ability to transfer resources between tenants.

Move a Resource

Verb	URI	Description
POST	/*`version`*/*`tenantId`*/path/to/resource/action/move?*`dest`*=`tenantId`	Move a Resource

Normal Response Code(s): 303, 301

Error Response Code(s): 404, 410, others …

A **POST** operation on a move action URL of a resource (…/path/to/resource/action/move) causes the resource specified by the path to move to the tenant in the *`dest`* URL parameter. The operation does not require a content body. On success, the service should return a 303 (See Other) with a `Location` header pointing to the resource's new home. The service should respond with a 404 (Not Found) if the resource does not exist or 410 (Gone) if the resource has been recently deleted. Additionally, a service may respond with a 301 (Moved Permanently) if the resource has already been moved. In this case, the `Location` header should point to the move action URL in the new resource location.

After the resource has been moved a service may respond with either a 404 (Not Found) or a 301 (Moved Permanently) to a **GET** request on the resource itself (…/path/to/resource). The 301 response must contain a `Location` header with an URL pointing to the resource's new location.

Move all Resources

Verb	URI	Description
POST	/*version*/*tenantId*/action/move?*dest*=tenantId	Moves all resources into a destination tenant.

Normal Response Code(s): 204, 202

Error Response Code(s): 404, 410, others …

A **POST** operation on a move action URL of a tenant (/*version*/*tenantId*/action/move) causes all resources in that tenant to move to the tenant specified by the *dest* URL parameter. This operation is very similar to the operation described above, except that it moves all resources in the tenant instead of a single resource. It is important to note that the tenant **MUST NOT** be deleted automatically after resources have been moved. Instead, an operator must explicitly issue a **DELETE** on the tenant. On success, the call should return a 303 (See Other) with a `Location` header pointing to the destination tenant. The service should respond with a 404 (Not Found) if the tenant does not exist or 410 (Gone) if the tenant has been recently deleted. A service may respond with either a 404 (Not Found) or a 301 (Moved Permanently) on a **GET** request on a previously moved resource. The 301 response must contain a `Location` header with an URL pointing to the resource's location in the new tenant.

Ensuring Consistency

The move operations above assume that resources are logically, and not physically, organized into tenants. In this case, move operations are virtual and can occur without the need to ensure consistency between resources as they move from one tenant to another. There may be cases, however, where tenants provide a physical organization of resources. For example, tenants may be placed in different service tiers and the tiers may be distributed among different sets of nodes in a cluster. Here, resources must be physically moved from one node to another, and operators must be assured that a resource is in a consistent state before it can be moved. The operations below allow for consistent moves by utilizing a *move action* resource.

Get a Move Action

Verb	URI	Description
GET	/*version*/*tenantId*/path/to/resource/action/move?*dest*=tenantId	Get resource move action.
GET	/*version*/*tenantId*/action/move?*dest*=tenantId	Get all resource move action.

Normal Response Code(s): 200, 203

Error Response Code(s): 404, 410, others …

A move action helps coordinate states as resources are moved from one tenant to another. Move actions must be acquired in cases where operators wish to ensure consistency between moves. An operator acquires a move action by performing a **GET** on the move action URL of either a specific resource (…/path/to/resource/action/move) or of an entire tenant (/*version*/*tenantId*/action/move). The destination tenant of the move must be specified in the *dest* URL parameter. An example request is illustrated below.

Example 3. Get Move Action Request

```
GET /v1.0/17776666/action/move?dest=176625343 HTTP/1.1
Host: service.openstack.com
```

Example 4. Get Move Action Response (Full)

```
HTTP/1.1 200 Okay
Date: Mon, 12 Nov 2010 15:55:01 GMT
Content-Type: application/xml; charset=UTF-8
ETag: "d8a5179a69519b32de12cad41705edd694790ffc"
```

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

<move xmlns="http://service.openstack.com/actions"
      dest="176625343">
    <tenants>
        <tenant id="17776666">
            .
            .
            .
        </tenant>
        <tenant id="176625343">
            .
            .
            .
        </tenant>
    </tenants>
    <resources>
        <resource id="1">
            .
            .
            .
        </resource>
        <resource id="2">
            .
            .
            .
        </resource>
        .
        .
        .
    </resources>
</move>
```

The response to the move action request is service-specific. The purpose of the response is to allow operators to confirm resource state before a move is requested. Thus the response **MUST** contain information about the state of resources and tenants that are affected by the move. An entity tag (`Etag`) header **MUST** be included in the response. The header **MUST** contain a quoted opaque string that uniquely identifies the response. In the example above we use a SHA1 digest of the response text. There may be cases where the number of resources affected by the move is very large. In these cases, the response **SHOULD NOT** contain a list of all resources affected, but rather it **SHOULD** contain a tag that uniquely identifies the current state of the affected resources. The response **SHOULD** also contain metadata that is common to all resources affected by the move. This is illustrated in the example below.

Example 5. Get Move Action Response (Tagged)

```
HTTP/1.1 200 Okay
Date: Mon, 12 Nov 2010 15:55:01 GMT
Content-Type: application/xml; charset=UTF-8
ETag: "50d935685fc4d998e202f44694371875d4dfebb7"
```

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

<move xmlns="http://service.openstack.com/actions"
      dest="176625343">
    <tenants>
        <tenant id="17776666">
            .
            .
            .
        </tenant>
        <tenant id="176625343">
            .
            .
            .
        </tenant>
    </tenants>
    <resources tag="f152f9be36f69f0b162b32fe2beed8c61b99e69b"
               size="10000" total-usage="2.5TB" />
</move>
```

Note that the tag in the content of the message is different from the one supplied via the `ETag`. The `ETag` uniquely identifies the move action response. The tag in the content identifies the state of all of the resources affected. Conceptually, one can think of it as the sum of all of the `ETag`s of the affected resources. It is also important to note that a change in the tag will cause the `ETag` to change.

On success, a request for a move action should return a response code of 200 (Okay) or 203 (Non-Authoritative Information) if the request is cached. Services should respond with a 404 (Not Found) if the tenant or resources does not exist. A return code of 410 (Gone) signifies that the tenant has been recently deleted.

Conditional Move

Verb	URI	Description
POST	/*version*/*tenantId*/path/to/resource/action/move?*dest*=tenantId	Perform a conditional move operation on a resource.
POST	/*version*/*tenantId*/action/move?*dest*=tenantId	Perform a conditional move operation on all resources.

Normal Response Code(s): 200, 203

Error Response Code(s): 404, 410, 412, others …

Conditional moves work exactly like unconditional move requests except that an `If-Match` header should be included containing the `ETag` of the move action. An example request is illustrated below.

Example 6. Conditional Move Request

```
POST /v1.0/17776666/action/move?dest=176625343 HTTP/1.1
Host: service.openstack.com
If-Match: "d8a5179a69519b32de12cad41705edd694790ffc"
Content-Type: application/xml
```

Here the move should fail with a 412 (Precondition Failed) if any change in state has occurred between **GET** request and the **POST** request.

Questions and Answers

1. Why go through the trouble of obtaining a move action? Why not simply fail a move request if a resource is in an unmovable state?

If a service can detect an unmovable state then it should certainly fail the move operation. That said, whether or not a resource is movable depends on the specific deployment. For example, an operator may have a rule that tenants are only allowed to have 100 resources. The move action request allows operators to enforce the rule on moves.

References

[1] T Berners-Lee. L Masinter. M McCahill. *Uniform Resource Locators (URL).* http://tools.ietf.org/html/rfc1738 .