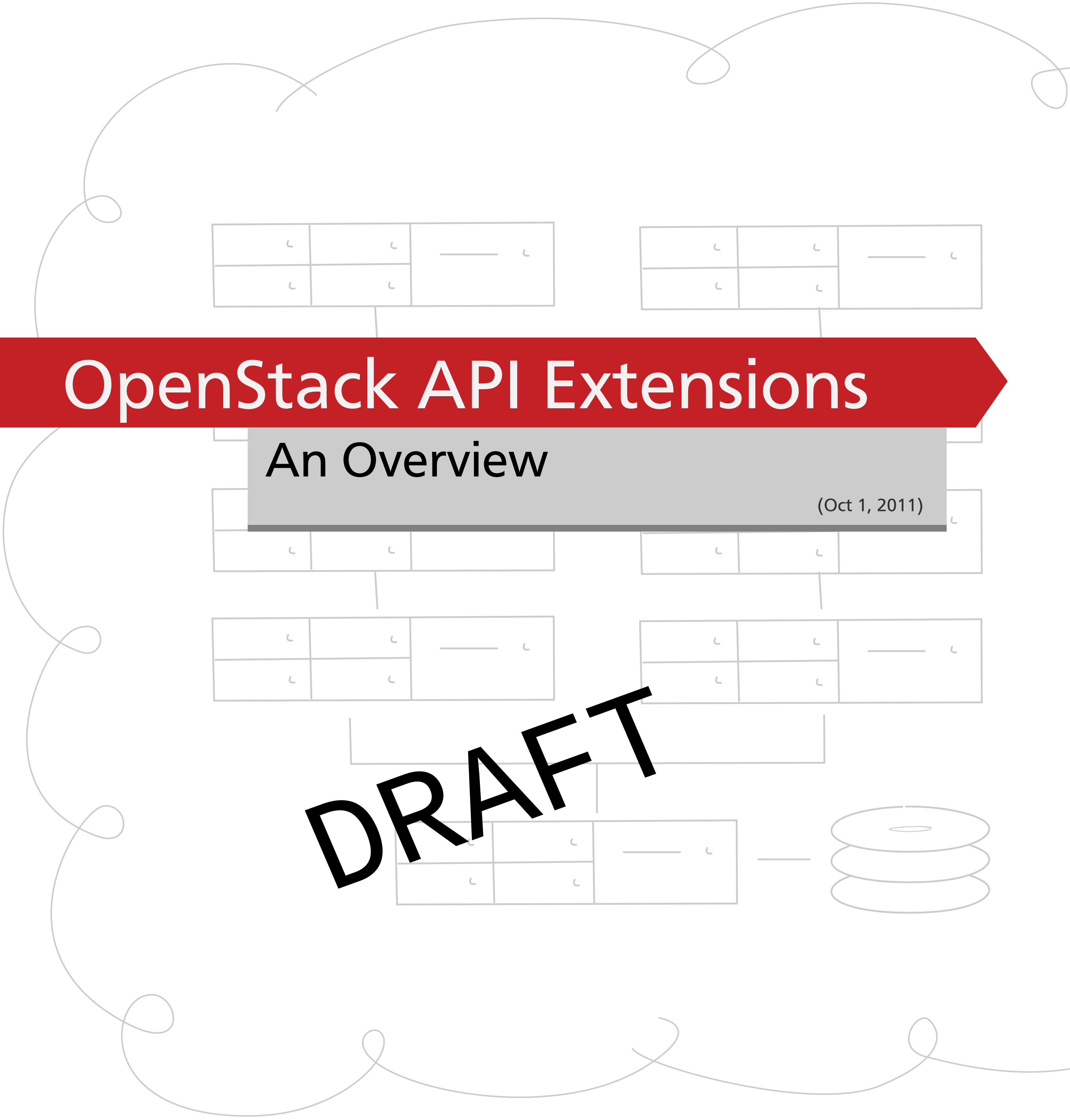


OpenStack API Extensions

An Overview

(Oct 1, 2011)

openstack™

docs.openstack.org

OpenStack API Extensions: An Overview

This document provides an overview of the OpenStack API extension mechanism.

Table of Contents

List of Figures

List of Tables

List of Examples

1. Overview

The OpenStack extension mechanism makes it possible to add functionality to OpenStack APIs in a manner that ensures compatibility with existing clients. This capability allows OpenStack operators and vendors to provide innovative functionality to their clients and provides a means by which new features may be considered in upcoming versions of OpenStack APIs.

This document describes the extension mechanism in detail. It provides guidance to API implementors and clients on developing and consuming API extensions, it describes the rules by which extensions are governed, and it describes the process used to promote API extensions to new features.

Warning

Please note that this document is a **draft**. It is intended to give developers an opportunity to provide feedback. If you begin implementing against this early access specification, please recognize that it is still subject to change. Comments, feedback, and bug reports are always welcomed; please submit these in the webhelp discussion forum at: http://docs.openstack.org/.

1.1. Intended Audience

This document is intended for software developers who wish either to implement or to consume an extendable OpenStack API. It assumes that the reader has a general understanding of:

- ReSTful web services
- HTTP/1.1
- JSON and/or XML data serialization formats
- At least one OpenStack API: Compute, Object Storage, etc.

1.2. Organization of this Document

Chapter 2, *Background*
: Provides background information on OpenStack extensions and the OpenStack extension mechanism.

Chapter 3, *Extensions and REST*
: Describes the specifics of implementing and consuming extensions in REST APIs.

Chapter 4, *Extension Governance and Promotion*
: Describes extension governance and the process by which extensions can be promoted to new features in future revisions of an API.

Chapter 5, *Summary*
: Briefly summarizes the benefits of the extension mechanism and provides a brief overview of how extensions are used.

1.3. Document Change History

Revision Date	Summary of Changes
Octobel 1, 2011	• Updated to reflect latest OpenStack governance model.
June 10, 2011	• Initial draft.

2. Background

This chapter provides background information on OpenStack extensions and the OpenStack extension mechanism. It describes what extensions are, how the extension mechanism in OpenStack is related to the OpenGL extension mechanism, the differences between extensions and versions, the concept of versioning extensions, and why extensions are vital when defining a pluggable architecture.

2.1. What are Extensions?

OpenStack APIs are defined strictly in two forms: a human-readable specification (usually in the form of a developer's guide) and a machine-processable WADL. These specifications define the core actions, capabilities, and media-types of the API. A client can always depend on the availability of this *core API* and implementers are always required to support it in its entirety. Requiring strict adherence to the core API allows clients to rely upon a minimal level of functionality when interacting with multiple implementations of the same API.

Note that it is quite possible that distinct implementations of an OpenStack API exist. First because API specifications are released under a free license, so anyone may use them to implement a core API. Furthermore, the OpenStack implementations themselves are released under a free license, making it possible to alter the code to create a specialized version. Such a specialized implementation could remain OpenStack-compatible even if it were to implement new features or add new capabilities, but only if it made the changes in a manner that ensures that a client expecting a core API would continue to function normally — this is where extensions come in.

An *extension* adds capabilities to an API beyond those defined in the core. The introduction of new features, MIME types, actions, states, headers, parameters, and resources can all be accomplished by means of extensions to the core API. In order for extensions to work, the core API must be written in such a manner that it allows for extensibility. Additionally, care should be taken to ensure that extensions defined by different implementers don't clash with one another, that clients can detect the presence of extensions via a standard method, and that there is a clear promotion path at the end of which an extension may become part of a future version of the core API. These actions, rules, and processes together form the *extension mechanism*. It is important that core APIs adhere to this mechanism in order to ensure compatibility as new extensions are defined. Note also that while a core API may be written to allow for extensibility, the extensions themselves are never part of the core.

2.2. Relationship to OpenGL

In the 1990s, OpenGL was developed as a portable open graphics library standard. The goal was to provide a cross-platform library that could enable developers to produce 3D graphics at real time speeds (30-120 frames per second). There were several major challenges to meeting this goal. In order to be considered an open standard, control needed to shift from Silicon Graphics (SGI), who originally developed OpenGL, to an independent Architecture Review Board (ARB) who would be responsible for approving specification changes, marking new releases, and ensuring conformance testing. Additionally, the graphics library itself would need to be designed in a manner that would allow the establishment of a stable and portable platform for developers. Finally, the library would need to garner the support of graphics hardware vendors as they would be providing the hardware acceleration needed to meet the goal of performing at real-time speeds.

Gaining vendor support is challenging because vendors are often in direct competition with one another. They differentiate themselves by creating innovative new features and by providing niche functionality to their users. Thus, OpenGL was faced with two competing requirements. On the one hand, it needed to abstract away vendor differences in order to provide a stable cross-platform environment to developers. On the other hand, in order to garner vendor support, it needed a method by which vendors could differentiate themselves and provide innovative new features and niche functionality to their users.

The OpenGL extension mechanism was developed to solve these problems. The extension mechanism achieved balance between the two requirements by maintaining the core specification under the direction of the Architecture Review Board while allowing vendors to define extensions to the core OpenGL specification. The core specification remained uncluttered and presented a unified view of common functionality. Because extensions were detectable at run time, developers could write portable applications that could adapt to the hardware on which they were running. This method of allowing for an extensible API has proven to be a very successful strategy. More than 500 extensions have been defined in OpenGL's lifetime and many vendors, including NVidia, ATI, Apple, IBM, and Intel, have participated in the process by developing their own custom extensions. Additionally, many key innovations (such as vertex and fragment shaders) have been developed via the extension process and are now part of the core OpenGL API.

OpenStack, while very different from OpenGL, shares many similar goals and faces many of the same challenges. OpenStack APIs are designed to be open API standards. An important goal is to provide developers with a ubiquitous, stable, any-scale cloud development platform that abstracts away many of the differences between hosting providers and their underlying infrastructure (hypervisors, load balancers, etc.). A Project Policy Board, similar to OpenGL's Architecture Review Board, is responsible for directing development of these APIs in a manner that ensures these goals are met. As with OpenGL, OpenStack requires support from vendors (and cloud providers) in order to be successful. As a result, OpenStack APIs also aim to provide vendors with a platform which allows them to differentiate themselves by providing innovative new features and niche functionality to their users. Because of these similarities, the OpenStack extension mechanism described in this document is modeled after the OpenGL extension mechanism. The methods by which extensions are defined vary drastically, of course, since the nature of the APIs is very different (C versus ReST); however, the manner in which extensions are documented, the way in which vendors are attributed, and the promotion path that an extension follows, all borrow heavily from OpenGL.

2.3. Extensions and Versions

Extensions are always interpreted in relation to a version of the core API. In other words, from a client's perspective, an extension modifies *a particular version* of the core API in some way. In reality, an extension may be applicable to several versions of an API at once. For example, a particular extension may continue to be available as a core API moves from one version to another. In fact, different implementations may decide to include support for an extension at different versions. As explained in Chapter 4, *Extension Governance and Promotion*, when an extension is defined, the minimal version of the core API that is required to run the extension is specified; implementers are free to support the extension in that version or in a later version of the core. Note, however, that because the extension mechanism allows for promotion, an extension in one version of a core API may become a standard feature in a later version.

Note

As always, implementers are not required to support an extension unless it is promoted to the core.

Because several versions of the core API may be supported simultaneously, and because each version may offer support for a different set of extensions, clients must be able to detect what versions and extensions are available in a particular deployment. Thus, both extensions and versions are queryable. Issuing a **GET** on the base URL (`/`) of the API endpoint returns information about what versions are available. Similarly, issuing a **GET** on the API's extensions resource (`/v1.1/extensions`) returns information about what extensions are available. (See Chapter 3, *Extensions and REST* for details of such requests.) Note that, since extensions modify a particular version of the API, the `extensions` resource itself is always accessed at a particular version.

Backward-compatible changes in an API usually require a minor version bump. In an extensible API, however, these changes can be brought in as extensions. The net effect is that versions change infrequently and thus provide a stable platform on which to develop. The Project Technical Lead (PTL), with the help of the OpenStack community, is responsible for ensuring that this stability is maintained by closely guarding core API versions. Extensions, however, can be developed without the consent or approval of the PRB. They can be developed in a completely decentralized manner both by individual OpenStack developers and by commercial vendors. Because extensions can be promoted to standard features, the development of new versions can be influenced significantly by individual developers and the OpenStack client community and is therefore not strictly defined by the PTL. In other words, new features of a core API may be developed in a bottom-up fashion.

That said, not all extensions are destined to be promoted to the next API version. Core APIs always deals with core functionality — functionality that is supported by all implementations and is applicable in common cases. Extensions that deal with niche functionality should always remain extensions.

The table below summarizes the differences between versions and extensions.

Table 2.1. Versions versus Extensions

Versions	Extensions
Rare. Versions provide a stable platform on which to develop.	**Frequent.** Extensions bring new features to the market quickly and in a compatible manner.
Centralized. Versions are maintained by the entity that controls the API Spec: the OpenStack Policy Review Board. Only the PRB can create a new version; only the PRB defines what "OpenStack Compute 1.1" means.	**Decentralized.** Extensions are maintained by third parties, including individual OpenStack developers and software vendors. Anyone can create an extension.
Core. Versions support core functionality.	**Niche.** Extensions provide specialized functionality.
Queryable. Issuing a **GET** on the base URL (`/`) of the API endpoint returns information about what versions are available.	**Queryable.** Issuing a **GET** on the API's extensions resource (`/v1.1/extensions`) returns information about what extensions are available.

2.4. Versioning Extensions

There is no explicit versioning mechanism for extensions. Nonetheless, there may be cases in which a developer decides to update an extension after the extension has been released and client support for the extension has been established. In these cases, it is recommended that a new extension be created. The extension may have a name that signifies its relationship to the previous version. For example, a developer may append an integer to the extension name to signify that one extension updates another: RAX-BAK2 updates RAX-BAK.

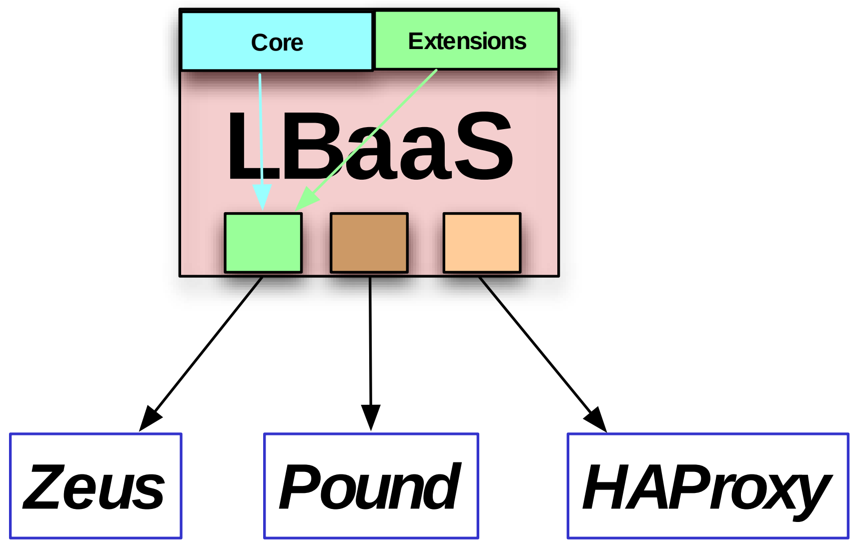
Extensions may have dependencies on other extensions. For example, RAX-BAK2 may depend on RAX-BAK and may simply add additional capabilities to it. In general, dependencies of this kind are discouraged and implementers should strive to keep extensions independent. That said, extension dependencies allow for the possibility of providing updates to existing extensions even if the original extension is under the control of a different vendor. This is particularly useful in cases where an existing extension has good client support.

2.5. Extensions and Pluggability

Core APIs abstract away vendor differences in order to provide a cross-platform environment to their clients. For example, a client should be able to interact with an OpenStack load balancing service without worrying about whether the deployment utilizes Zeus, Pound, or HAProxy on the backend. OpenStack implementations strive to support multiple backends out of the box. They do so by employing software drivers. Each driver is responsible for communicating to a specific backend and is in charge of translating core API requests to it.

The core API contains only those capabilities which are applicable to all backends; however, not all backends are created equal, with each backend offering a distinct set of capabilities. Extensions play a critical role in exposing these capabilities to clients. This is illustrated below. Here, extensions fill in the gap between the common capabilities that the core API provides and the unique capabilities of the Zeus load balancer. In a sense, one can think of extensions as providing frontends to OpenStack plug-ins.

Figure 2.1. Extensions and Pluggability



3. Extensions and REST

This chapter describes how extensions are implemented and consumed in REST APIs.

3.1. Identifying Extensions

Extensions are queryable. Clients can detect what extensions are available for any specific version of an OpenStack API. These extension queries return metadata about the extension including details on who created the extension and documentation on how the extension works.

The following operations must be supported by all OpenStack APIs:

Verb	URI	Description
GET	`/v1/extensions`	Returns a list of all available extensions.
GET	`/v1/extensions/{alias}`	Lists information about a particular extension if it's available.

3.1.1. List Extensions

Verb	URI	Description
GET	`/v1/extensions`	Returns a list of all available extensions.

Normal Response Code(s): 200, 203

This operation returns a complete list of all extensions available to an OpenStack API. It is important to note that extensions are always accessed at a particular version. In this case, the extensions listed in the response modify version **v1** of a specific OpenStack API. Extensions that modify **v2** of the API may be accessed at `/v2/extensions`. If the MIME type versioning scheme is used, instead of the URI scheme, then extensions will always be found in `/extensions`.

An implementation may support a large number of extensions. Because of this, the extension list may be paginated via a `next` link.

This operation does not require a request body.

Example 3.1. List Extensions Request: XML

```
<?xml version="1.0" encoding="UTF-8"?>
<extensions xmlns="http://docs.openstack.org/common/api/v1.0"
            xmlns:xs="http://www.w3.org/2001/XMLSchema"
            xmlns:atom="http://www.w3.org/2005/Atom"
            xmlns:osx="http://docs.openstack.org/common/api/xsl/v1.0">
    <extension name="Disk Configuration Extension"
              namespace="http://docs.rackspacecloud.com/servers/api/ext/
diskConfig/v1.0"
              alias="RAX-DCF"
              updated="2011-09-27T00:00:00">
        <description>
            Adds support for disk management flag on servers and images.
        </description>
        <atom:link rel="describedby" type="application/pdf"
                 href="http://docs.rackspacecloud.com/servers/api/ext/rax-dcf.
pdf"/>
        <atom:link rel="describedby" type="application/xml"
                 href="http://docs.rackspacecloud.com/servers/api/ext/rax-dcf/
api.xsd"/>
    </extension>
    <extension name="Backup Schedule Extension"
              namespace="http://docs.rackspace.com/servers/api/ext/backup/v1.
0"
              alias="RAX-BAK"
              updated="2011-09-12T10:40:00-04:00">
        <description>
            Allows the schedule of periodic (daily and
            weekly) images from a server.
        </description>
        <atom:link rel="describedby" type="application/pdf"
                 href="http://docs.rackspacecloud.com/servers/api/ext/rax-
backup.pdf"/>
        <atom:link rel="describedby" type="application/xml"
                 href="http://docs.rackspacecloud.com/servers/api/ext/rax-
backup/rax-backup.wadl"/>
    </extension>
</extensions>
```

Example 3.2. List Extensions Request: JSON

```
{
    "extensions": [
        {
            "name": "Disk Configuration Extension",
            "namespace": "http://docs.rackspacecloud.com/servers/api/ext/
diskConfig/v1.0",
            "alias": "RAX-DCF",
            "updated": "2011-09-27T00:00:00",
            "description": "Adds support for disk management flag on servers
 and images.",
            "links": [
                {
                    "rel": "describedby",
                    "type": "application/pdf",
                    "href": "http://docs.rackspacecloud.com/servers/api/ext/
rax-dcf.pdf"
                },
                {
                    "rel": "describedby",
                    "type": "application/xml",
                    "href": "http://docs.rackspacecloud.com/servers/api/ext/
rax-dcf/api.xsd"
                }
            ]
        },
        {
            "name": "Backup Schedule Extension",
            "namespace": "http://docs.rackspace.com/servers/api/ext/backup/v1.
0",
            "alias": "RAX-BAK",
            "updated": "2011-09-12T10:40:00-04:00",
            "description": "Allows the schedule of periodic (daily and weekly)
 images from a server.",
            "links": [
                {
                    "rel": "describedby",
                    "type": "application/pdf",
                    "href": "http://docs.rackspacecloud.com/servers/api/ext/
rax-backup.pdf"
                },
                {
                    "rel": "describedby",
                    "type": "application/xml",
                    "href": "http://docs.rackspacecloud.com/servers/api/ext/
rax-backup/rax-backup.wadl"
                }
            ]
        }
    ]
}
```

3.1.2. Get Extension

Verb	URI	Description
GET	`/v1/extensions/{alias}`	Lists information about a particular extension if it's available.

Normal Response Code(s): 200, 203

Error Response Code(s): 404

This operation returns basic information about an extension. Again, note that the extension is accessed at a particular version. In this case, the extension extends version **v1** of an OpenStack API.

The alias uniquely identifies an extension and contains a vendor prefix, in the example below RAX, that prevents clashes between extensions written by different vendors. See Chapter 4, *Extension Governance and Promotion* for a detailed description of vendor prefixes. A 404 is returned if the extension is not supported, so this resource can be used to quickly check for the presence of a particular extension.

Table 3.1. Get Extension Request Parameters

Name	Style	Type	Description
`alias`	template	String	The extensions alias uniquely identifies an extension.

This operation does not require a request body.

Example 3.3. Get Extension Request: XML

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

<extension xmlns="http://docs.openstack.org/common/api/v1.0"
           xmlns:atom="http://www.w3.org/2005/Atom"
           name="Backup Schedule Extension"
           namespace="http://docs.rackspace.com/servers/api/ext/backup/v1.0"
           alias="RAX-BAK"
           updated="2011-09-12T10:40:00-04:00">
    <description>
        Allows the schedule of periodic (daily and
        weekly) images from a server.
    </description>
    <atom:link rel="describedby" type="application/pdf"
               href="http://docs.rackspacecloud.com/servers/api/ext/rax-
backup.pdf"/>
    <atom:link rel="describedby" type="application/xml"
               href="http://docs.rackspacecloud.com/servers/api/ext/rax-
backup/rax-backup.wadl"/>
</extension>
```

Example 3.4. Get Extension Request: JSON

```
{
    "extension": {
        "name": "Backup Schedule Extension",
        "namespace": "http://docs.rackspace.com/servers/api/ext/backup/v1.0",
        "alias": "RAX-BAK",
        "updated": "2011-09-12T10:40:00-04:00",
        "description": "Allows the schedule of periodic (daily and weekly)
 images from a server.",
        "links": [
            {
                "rel": "describedby",
                "type": "application/pdf",
                "href": "http://docs.rackspacecloud.com/servers/api/ext/rax-
backup.pdf"
            },
            {
                "rel": "describedby",
                "type": "application/xml",
                "href": "http://docs.rackspacecloud.com/servers/api/ext/rax-
backup/rax-backup.wadl"
            }
        ]
    }
}
```

3.1.3. Extension Metadata

The extension queries described above return basic metadata about an API's extensions. Clients can depend on the metadata described in the following example.

Example 3.5. Extension Metadata: XML

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

<extension xmlns="http://docs.openstack.org/common/api/v1.0"
           xmlns:atom="http://www.w3.org/2005/Atom"
  ❶         name="Backup Schedule Extension"
  ❷         namespace="http://docs.rackspace.com/servers/api/ext/backup/v1.0"
  ❸         alias="RAX-BAK"
  ❹         updated="2011-09-12T10:40:00-04:00">
    <description>
  ❺      Allows the schedule of periodic (daily and
        weekly) images from a server.
    </description>
  ❻  <atom:link rel="describedby" type="application/pdf"
               href="http://docs.rackspacecloud.com/servers/api/ext/rax-
backup.pdf"/>
    <atom:link rel="describedby" type="application/xml"
               href="http://docs.rackspacecloud.com/servers/api/ext/rax-
backup/rax-backup.wadl"/>
</extension>
```

Example 3.6. Extension Metadata: JSON

```
{
    "extension": {
  ❶      "name": "Backup Schedule Extension",
  ❷      "namespace": "http://docs.rackspace.com/servers/api/ext/backup/v1.0",
  ❸      "alias": "RAX-BAK",
  ❹      "updated": "2011-09-12T10:40:00-04:00",
  ❺      "description": "Allows the schedule of periodic (daily and weekly)
 images from a server.",
  ❻      "links": [
            {
                "rel": "describedby",
                "type": "application/pdf",
                "href": "http://docs.rackspacecloud.com/servers/api/ext/rax-
backup.pdf"
            },
            {
                "rel": "describedby",
                "type": "application/xml",
                "href": "http://docs.rackspacecloud.com/servers/api/ext/rax-
backup/rax-backup.wadl"
            }
        ]
    }
}
```

❶ A human readable name assigned to the extension.

❷ A namespace that uniquely identifies the extension. The namespace should be used when extending XML media types.

❸ A short name that uniquely identifies the extension. Extension aliases are prefixed with a vendor identifier, here RAX, to prevent clashes between vendors. Extension aliases are used when extending non-XML media types, headers, parameters, resources, actions, and states.

❹ A timestamp indicating when the extension was last updated. The timestamp can be used to monitor changes to an extension while it's in development.

❺ A human readable description of the extension.

❻ A collection of `describedby` links that provide further details on the extension. At least on link with a media type of `application/pdf` is required — these PDF links should point to a detailed human readable specification of the extension. Other `despriedby` Links are also allowed, including links to WADLs and XSD and JSON Schema files.

3.2. What can be extended and how

Extensions may define:

- **New elements and attributes.** Extensions may add additional data elements or attributes to existing MIME types.
- **New resources.** Extensions may define entirely new API resources.
- **New parameters.** Extensions may introduce new query parameters to existing resources.
- **New headers.** Extensions may define new HTTP headers.
- **New verbs.** Extensions may introduce support for an HTTP verb (**PUT**, **POST**) to an existing core resource.
- **New media types.** Extensions may define entirely new representation types for existing resources.
- **New actions.** Extensions may define new actions on existing resources.
- **New states.** Extensions may introduce new states in an API state machine.
- **Other capabilities.** Extensions may define other general API capabilities. For example, the ability to edit an otherwise uneditable attribute.

3.2.1. Adding new elements and attributes

Extensions may add additional data elements or attributes to existing MIME types. In XML, attributes may be added to elements so long as they in the extension namespace. New elements must also be in the extension namespace and must be added after the core elements.

Example 3.7. Adding a new attribute: XML

```
<image xmlns="http://docs.rackspacecloud.com/servers/api/v1.0"
    xmlns:RAX-PIE="http://docs.rackspacecloud.com/servers/api/ext/pie/v1.0"
    id="1" name="CentOS 5.2"
    serverId="12"
    updated="2010-10-10T12:00:00Z"
    created="2010-08-10T12:00:00Z"
    status="ACTIVE"
    RAX-PIE:shared="true"
/>
```

Example 3.8. Adding a new element: XML

```
<image xmlns="http://docs.rackspacecloud.com/servers/api/v1.0"
    xmlns:RAX-DESC="http://docs.rackspacecloud.com/servers/api/ext/desc/v1.0"
    id="1" name="CentOS 5.2"
    serverId="12"
    updated="2010-10-10T12:00:00Z"
    created="2010-08-10T12:00:00Z"
    status="ACTIVE">
    <RAX-DESC:description>
        This is a sample image description.
    </RAX-DESC:description>
</image>
```

New attributes and elements may also be added in JSON. Both of these cases equate to adding a new property in JSON. Here, the property name should be prefixed by the alias and a colon as illustrated below.

Example 3.9. Adding a new properties in JSON

```
{
    "image" : {
        "id" : 1,
        "name" : "CentOS 5.2",
        "serverId" : 12,
        "updated" : "2010-10-10T12:00:00Z",
        "created" : "2010-08-10T12:00:00Z",
        "status" : "ACTIVE",
        "RAX-PIE:shared" : true
    }
}
```

```
{
    "image" : {
        "id" : 1,
        "name" : "CentOS 5.2",
        "serverId" : 12,
        "updated" : "2010-10-10T12:00:00Z",
        "created" : "2010-08-10T12:00:00Z",
        "status" : "ACTIVE",
        "RAX-DESC:description" : "This is a sample image description."
    }
}
```

3.2.2. Adding new resources

Extensions may define entirely new API resources. These resources should be offset by the extension alias. For example `/RAX-BAK/backup-schedule`. This alias may occur at any part of the path: `/v1.1/39923/servers/3345/RAX-BAK/backup-schedule`. When an extension is promoted into the core the extension alias, here `RAX-BAK`, will simply be dropped. A vendor may define multiple resources within the same alias.

3.2.3. Adding new parameters

Extensions may introduce new query parameters to existing resources. The parameter name should be prepended by the extension alias followed by a colon. For example, a **GET** against https://api.servers.rackspacecloud.com/v1.0/224532/servers?RAX-PIE:img=1244 sets the `RAX-PIE` extension's *`img`* parameter to the value of 1244. The extension alias and colon is dropped from the parameter when the extension is promoted into the core API.

3.2.4. Adding new headers

Extensions may define new HTTP headers. The header names should be be prepended by a unique extensions alias followed by a dash. In the example below, `X-Auth-User` and `X-Auth-Key` relate to the core API; `RAX-CBS-Header1` and `RAX-CBS-Header2` relate to the API's RAX-CBS extension. Here too, the extension alias and the dash is dropped when an extension is promoted into core.

Example 3.10. Authentication Request with Extended Headers

```
GET /v1.0 HTTP/1.1
Host: auth.api.openstack.com
X-Auth-User: jdoe
X-Auth-Key: a86850deb2742ec3cb41518e26aa2d89
RAX-CBS-Header1: value1
RAX-CBS-Header2: value2
```

3.2.5. Adding support for a new HTTP verb

An extension may introduce support for an otherwise unsupported HTTP verb (**PUT**, **POST**) to an existing core resource. To avoid clashes with other extensions, the extension may introduce the additional verb to a URI other than the one being extended. For example, assume that an extension (`RAX-PATCH`) adds support for **PATCH** to a URI to an image `/v1.1/1234/image/1`. The extension may allow the **PATCH** verb on a separate URI (using the rules in Section 3.2.2, “Adding new resources”), `/v1.1/1234/image/RAX-PATCH/1`. The extension should add a link to the image representation so that the URI is easily discoverable. Note that the unique extension alias is used as the link relation type in this case.

Example 3.11. Discoverable PATCH URI: XML

```
<image xmlns="http://docs.rackspacecloud.com/servers/api/v1.0"
    xmlns:atom="http://www.w3.org/2005/Atom"
    id="1" name="CentOS 5.2"
    serverId="12"
    updated="2010-10-10T12:00:00Z"
    created="2010-08-10T12:00:00Z"
    status="ACTIVE">
    <atom:link
        rel="RAX-PATCH"
        href="http://servers.api.openstack.org/1234/image/RAX-PATCH/1"/>
</image>
```

Example 3.12. Discoverable PATCH URI: JSON

```
{
    "image" : {
        "id" : 1,
        "name" : "CentOS 5.2",
        "serverId" : 12,
        "updated" : "2010-10-10T12:00:00Z",
        "created" : "2010-08-10T12:00:00Z",
        "status" : "ACTIVE",
        "links": [
                {
                    "rel": "RAX-PATCH",
                    "href": "http://servers.api.openstack.org/1234/image/RAX-
PATCH/1"
                }
        ]
    }
}
```

3.2.6. Adding new media types

Extensions may define entirely new media types for existing resources. For example, an extension may introduce an Atom representation in addition to XML and JSON. Here the extension should simply utilize HTTP's existing content negotiation mechanism. For example, an extension that adds support for atom should allow clients to send an ACCEPT header with `application/atom+xml`.

It may be possibile for multiple extensions to add support for the same media type. The situation should generally be avoided. Ideally, both extensions may behave in the exact same way. In cases where a distinction need be made, clients should pass the unique extension prefix via the `ext` mime type parameter as a hint to an implementaion that a particular extension is requested. For example, `application/atom+xml;ext=RAX-ATOM`.

3.2.7. Adding new actions

The OpenStack Compute API defines operations on a server as Server Actions. Extensions may define additional actions. In XML, actions must be specified in the extension namespace.

Example 3.13. Extended Action: XML

```
<mount_volume xmlns="http://docs.rackspacecloud.com/servers/api/ext/cbs/v1.0"
    CBSID="123"/>
```

Actions should be prefixed in JSON by the extension alias followed by a colon, as illustrated below.

Example 3.14. Extended Action: JSON

```
{
    "RAX-CBS:mount_volume" : {
        "CBSID" : "123"
    }
}
```

3.2.8. Adding new states

The OpenStack Compute API also has the concept of a state machine. Extensions may introduce new states. These states should be prepended by the unique extension alias followed by a colon. The extened state transitions should be clearly documented by the extension. In the example below, a server may enter a `RAX-CBS:ATTACHING` state only from an ACTIVE state.

Example 3.15. Introducing a new state

```
ACTIVE → RAX-CBS:ATTACHING → ACTIVE
```

3.2.9. Adding other capabilities

Extensions may define other general API capabilities. For example, the ability to edit an otherwise uneditable attribute. The following general principles should be applied in these cases. First, care should be taken to avoid clashing with other extensions, the extension prefix or XML namespace should be used for this purpose. Next, the extension should be modeled such that the promotion path is well understood. And finally, the extensions should be well documented so that changes to the core API are well understood.

4. Extension Governance and Promotion

This chapter describes extension governance and the process by which extensions can be promoted to new features in future revisions of an API. It describes how to build extensions that will be likely promoted and it introduces the extension registry.

4.1. Specification Governance and OpenStack Extensions

API governance is shared between the project team lead (PTL) and the project policy board (PPB). The project team lead (PTL) handles concerns about functionality and resolves conflicts; the policy board provides oversight to maintain consistency among teams.

A PTL may designate an extension as an OpenStack approved extension. OpenStack deployments should offer support for OpenStack extensions as some of these exenensions may be on the fast track for inclusion as standard core features. When a PTL approves an extension, the extension's prefix is changed so that rather than identifying a specific vendor it identifies OpenStack as its source. OpenStack extesions are prefixed with `OS`. For example, if the `RAX-PIE` extension were to be promoted to an OpenStack extension, the approved version would be known as `OS-PIE`.

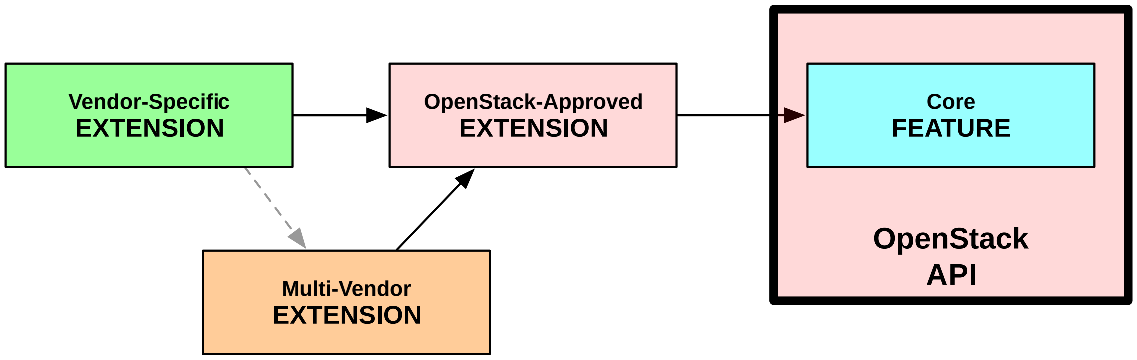
4.2. The Promotion Path

The simplest promotion path for an extension is from a vendor extension to an OpenStack extension to a core API feature.

Some extensions may be developed cooperatively by more than one vendor; these are known as Multi-Vendor extensions and are identified by the `MLTI` vendor prefix. Alternatively, an extension may begin with a single vendor but become a Multi-Vendor extension at some point in its life. In either of these cases, the extension follows same promotion path from Multi-Vendor status to OpenStack approval to implementation as a core API feature.

This multi-step promotion path allows competing extensions to coexist and makes it possible to experiment with new features before deciding whether to incorporate them into the API. It also means that API specifications are written bottom-up rather than top-down, so that implementations determine new features and the API is not designed in a vacuum.

Figure 4.1. An API extension can become a core feature of the API if approved by OpenStack.



Some extensions should never be promoted to core features. This can be because the extension implements niche functionality that doesn't make sense in the core API, or because it implements functionality that can't be used within the core API since it would prevent a particular backend from implementing the full set of core features.

For example, the ability to dynamically change the port number for a load balancer might be useful as an extension but problematic in the core API. If most but not all load balancers could support this feature, the extension must not be promoted into the core API. Instead, providing the feature via an API extension makes it available to customers who would like to use it but prevents it from creating problems for those who cannot use it.

4.3. Creating Extensions Likely to Be Promoted Into Core

Extensions characterized by consensus, stability, and standardization are most likely to be approved and promoted by a PTL. Multiple decisions and actions between the invention of a new extension and its approval by OpenStack relate to establishing broad support for the extension and preparing it to work consistently within the larger OpenStack community.

Extensions created by a single, independent author can be promoted to OpenStack-approved extensions and ultimately into the core API. However, an extension is most likely to be promoted if consensus has been established around it first. Multi-Vendor status is a strong indication that an extension is useful beyond a niche application; extensions identified by the `MLTI` prefix may have an advantage when a PTL considers which extensions to promote.

Before undertaking to create a new extension, vendors and individual software developers should compare their emerging ideas to existing extensions in the extension registy (see Section 4.4, “The Extension Registry”). If an extension already provides similar or related functionality, joining an ongoing effort to develop and enhance it may be a more productive approach than initiating a new effort to develop an independent extension. This can accomplish two key goals: individuals and groups with shared interests can pool their efforts to develop an extension that meets all their needs; the OpenStack community can recognize highly-valued extensions and support their approval and promotion, making those extensions widely available as quickly as possible.

Similarly, even if no existing extension is a good match for a new idea, development, approval, and promotion of a new extension can be facilitated by enlisting multiple vendors in the project. Just as OpenStack provides methods for software developers to communicate and collaborate on larger projects, it can enable formation of multi-vendor teams to create a new extension.

4.4. The Extension Registry

OpenStack maintains an extensions registry. The registry serves the following purposes:

1. It provides a central repository for all OpenStack extensions organized by API.
2. It allows vendors to register vendor prefixes: (RAX, HP, etc.).

Note that the registry is a repository of extension specifications and *not* a repository of extension software or plugins for a particular OpenStack implementation. That is, the registry is implementation agnostic and may describe extensions supported by implementations other than the OpenStack reference implementations (Nova, Glance, Swift, Keystone).

4.4.1. Benefits of a Centralized Registry

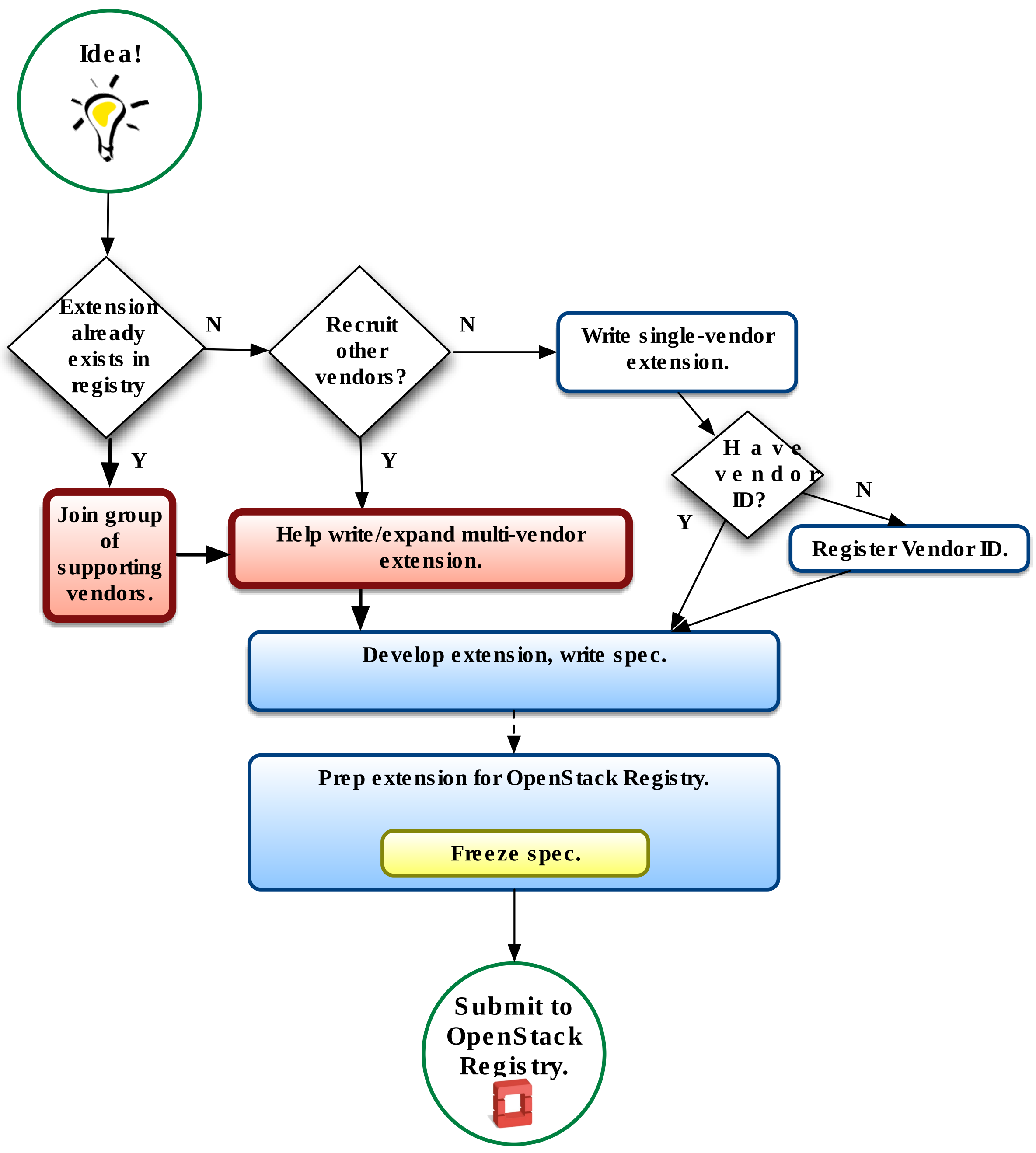
Users of OpenStack APIs benefit from having a centralized extension repository. While we would like to encourage vendors to develop extensions in coordination with the OpenStack community, some vendors may opt to develop their extensions independently. Without a centralized extension repository, users would have to gather information from various vendor websites in order to burst from one cloud onto another. Adding to the confusion is the fact that vendor X may choose to provide support for an extension developed by vendor Y. It may, therefore, not be clear to users where to go to get information about an extension.

Extension authors also benefit from having a central repository. The repository can be used to determine whether an existing extension satisfies an author's requirements. This can help mitigate the problem of extension proliferation and it may provide an opportunity for authors of similar extensions to combine their efforts. Additionally, the repository may provide a means by which users can become aware of extensions not currently supported by their vendors, thus allowing them to lobby for such support.

4.4.2. Extension Registration

The extension registration process closely follows the registration process developed by the Khronos group for OpenGL: http://www.opengl.org/registry/doc/rules.html. The process is illustrated at a high level in Figure 4.2.

Figure 4.2. The Extensions Registration Process


Idea!
Extension already exists in registry
N
Recruit other vendors?
N
Write single-vendor extension.
Y
Y
Have vendor ID?
N
Y
Join group of supporting vendors.
Help write/expand multi-vendor extension.
Register Vendor ID.
Develop extension, write spec.
Prep extension for OpenStack Registry.
Freeze spec.
Submit to OpenStack Registry.

When creating an extension the following steps should be followed:

1. **Determine if an existing extension can be used.** Reuse of an existing extension is encouraged as this helps prevent the unnecessary proliferation of extensions and provides a means by which popular extension can be promoted to the core. Additionally, it's possible that existing extensions have established client support.

   If an extension with similar functionality exists, but does not entirely meet the authors needs, then authors are encouraged to collaborate on promoting the extension into a multi-vendor extension.

   Finally, an extension may be under development but has not yet been released. Thus, authors are also encouraged to send an email to the OpenStack mailing list to see if anyone else is working on an extension with similar functionality.
2. **Determine if a multi-vendor or OpenStack extension is appropriate.** If a suitable extension does not exist, an author should work with other vendors to determine if a muliti-vendor extension is appropriate. As with single vendor extensions, multi-vendor extensions may be implemented by interested parties among themselves without coordination with the OpenStack community. That said, as stated in Section 4.3, multi-vendor extensions are preferred over single vendor extension during the promotion processes.

   If the extension provides useful functionality that is applicable to a wide variety of general circumstances, the author should consider developing the extension as an OpenStack extension. OpenStack extensions are the most "blessed" category of extensions. If they are generally applicable, these extensions are on the fast track to becoming core API features. Reference implementations will always offer support for these extensions and alternate implementations are also encouraged to support them. OpenStack extensions should be implemented in coordination with the OpenStack community and by working closely with the PTL of the service being extended. OpenStack extensions go through a standardization process to ensure that they integrate well with the core API.
3. **Register a vendor prefix, if one has not already been assigned.** If a single vendor extension is appropriate an author should register a vendor prefix with the extension registry. The vendor prefix is a short string that is used to prevent clashes between extensions. The following are examples of vendor prefixes:

Prefix	Vendor
RAX	Rackspace
HP	Hewlett-Packard
CTX	Citrix

   As stated in Chapter 3, *Extensions and REST*, extension aliases are always prefixed with these vendor identifiers. For example, RAX-BAK is the extension alias for Rackspace's OpenStack Compute Backup Schedule Extension. These aliases are used to prevent clashes in HTTP parameters, headers, and JSON representations. An example backup schedule request is illustrated below.

Example 4.1. RAX-BAK JSON Representation

```
{
    "RAX-BAK:backupSchedule" : {
        "enabled" : true,
        "weekly" : "THURSDAY",
        "daily" : "H_0400_0600"
    }
}
```

Any vendor or individual may register a vendor prefix by simply sending an email to extensions@openstack.com. There is no formal approval process for prefixes other that new prefixes are not allowed to clash with existing ones.

4. **Develop and write the extension specification.** Authors should start with a template for writing extension specifications. Templates exist in DocBook and reStructuredText formats. These templates should be available in the extension registry.

   Extensions must be written against a specific version of a core OpenStack API. If possible, it is highly preferable to write extensions against the most recent stable version of the API. "Written against" means that new language must be written as well-defined modifications to the core API being referenced. It is important for authors to document how the core API is affected by the introduction of the extension. For example: Will the extension introduce new states in the compute state machine? Should a new action be introduced to bring a server to an `ACTIVE` state? It should be possible for someone not involved with the development of an extension to sit down with a copy of the core API specification and the extension specification and produce a merged document that clearly specifies how the extended API should function.
5. **Freeze the specification and submit it to the registry.** When the extension specification is frozen, it should be submitted via email to extensions@openstack.com for inclusion in the extension registry. Vendor specifications will be published as is, though they may be edited to meet style an formatting guidelines. OpenStack extensions must gain the approval of the project's PTL before publication and may undergo a standardization process. It is expected that a dialog with the project's PTL and the OpenStack community be initiated before an OpenStack extension is submitted for publication.

   Note that an extension must be stable and its specification frozen before it can be considered for approval as an OpenStack extension. This makes it possible to consider a clear and finite definition of the extension. Otherwise, PTL approval would amount to a blank check, authorizing future development of the extension in unforeseen directions.
6. **Make the registry the source of truth for the specification.** Once published, vendors are encouraged to link directly to the extension registry when referencing the specification and to not have external copies of the specification that may fall out of sync with the registry. The OpenStack extension registry should be the source of truth for all frozen OpenStack extension specifications.