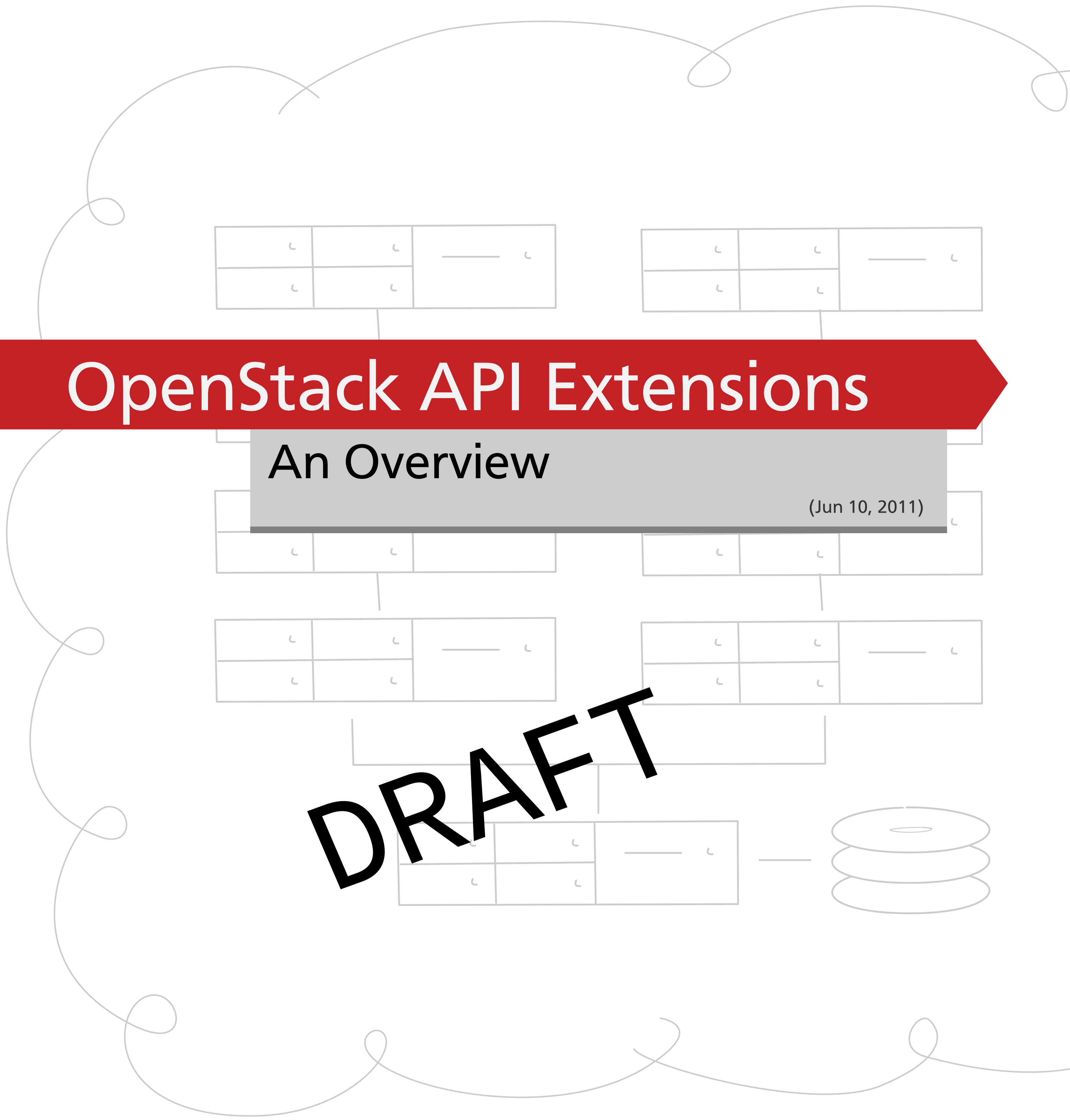


OpenStack API Extensions

An Overview

(Jun 10, 2011)

openstack™

docs.openstack.org

OpenStack API Extensions: An Overview

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

List of Figures

List of Tables

1. Overview

Table of Contents

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.

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.

Describes the specifics of implementing and consuming extensions in ReST APIs.

Describes API governance and the process by which extensions can be promoted to new features in future revisions of an API.

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
June 10, 2011	• Initial draft.

2. Background

Table of Contents

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 Policy Review 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 , 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 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 Policy Review Board (PRB), with the help of project team leaders, 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 PRB. 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-PIE2` updates `RAX-PIE`.

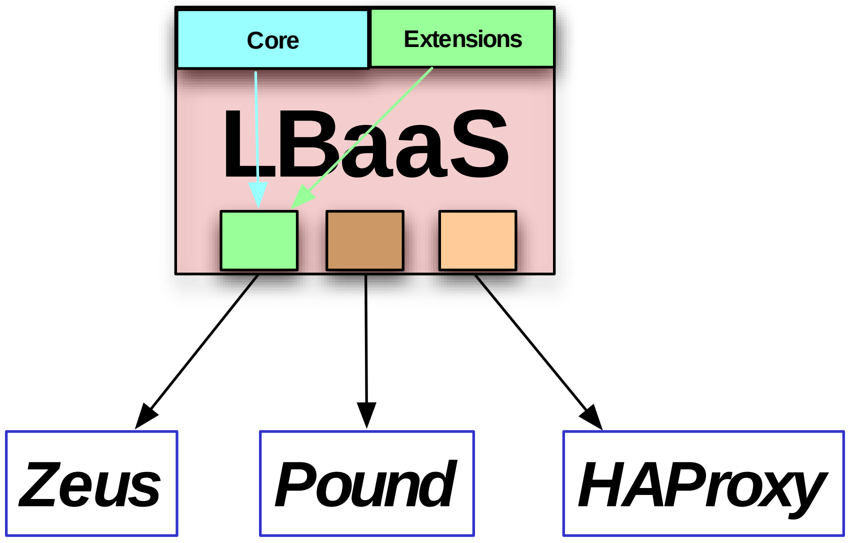
Extensions may have dependencies on other extensions. For example, `RAX-PIE2` may depend on `RAX-PIE` and may simply add additional capabilities to it. In general, dependencies of this kind are discouraged and implementers should strive to keep extensions independent. That said, extension dependencies 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


Core
Extensions
LBaaS
Zeus
Pound
HAProxy