LibcloudImages

The goal of this document is to provide all of the framework, and as much code as possible, to allow for a standard image format to exist on a variety of cloud hosting providers. There are two main components; 1) An image rendering service, 2) Required APIs that support the format

libcloud-images-diagram.png

Image rendering service

The image rendering service is responsible for downloading, caching, and preparing an image for booting on the provider specific infrastructure. It is structured around being a REST-style API, from which providers are abstracted away from most details about how images are built.

The image rendering service can be broken into several parts:

- Resource Fetching
- Resource Caching
- Image Conversion
- Image Output Storage

Each major area will have plugins to provide easy choice for providers, based on their own architectures, but at the same time, many plugins use cases will be shared between many providers.

Lets take the example of a user creating and booting a new Cloudlets image. The user calls the providers create_image call, with a JSON representation of their image. The provider validates the JSON and adds its own attributes, optionally overriding the Kernel or other attributes. This JSON repersentation is then sent to the Image Rendering endpoint.

The endpoint validates the JSON, and then returns a opaque ID for this image, and the provider must then store this association with its user. The service then begins fetching resources like a the Cloudlets tarball over HTTP, or fetching the data from another plugin backend like a distributed filesystem. Once the required resources are fetched either remotely or from a local cache, the image is converted into the providers preferred format, in this example a bootable ext3 image suitable for use by Xen.

Once the bootable image is created, it is stored by an output plugin in a provider specific fashion, which might be SAN, a remote HTTP url, or a simple file path.

During any point in this phase, the provider can ask the image rendering service about the status of the image using the opaque ID, and it will return what stage it is in, and how long it estimates it will take to complete.

The image rendering service uses a small SQL based database to store its own metadata information, but this is

Image Service APIs

List Images

URL: /images Method: GET Return Content Type: JSON

Returns a list of all images stored in the image rendering service.

Create Image

URL: /images Method: PUT Payload Type: JSON Return Content Type: JSON

A description of the image containing a URL from which to fetch resources to build the image. Once submitted, the image service takes care of conversion to the configured image type and output locations.

Location of the data Template variables Provider adds or overrides own template variables Private key for image decryption, optional SHA-1 checksum in the manifest, optional Data is a tarball hosted at a URL Cache data indefinitely Template types (js, google c template)

Image Status

URL: /images/${ID}
Method: GET Return Content Type: JSON

Returns the current status of an image. Once the image built, this includes the URL to the bootable location as returned by the output storage plugin.

Validate - Manifest is being validated Fetch - Tarball is being downloaded Build - Preparing the filesystem Render - Running the templating engine Store - Storing post-rendered image to storage Ready - Image is ready to be booted Delete - Image is being deleted 404 - if it does not exist

Fork Image

URL: /images/${ID}/fork Method: POST Payload Type: JSON Return Content Type: JSON

Re-runs with templates with the cached image, but with the payloads updated configuration variables. Merge existing manifest with updated variables from the payload. Returns a new image id.

On the backend: Take the filesystem, take the list of templated files and rerun them with the new arguments.

Use cases: Take a base image and update it with new passwords, ipaddresses, hostnames, etc

Delete Image

URL: /images/${ID}
Method: DELETE Return Content Type: JSON

Deletes the reference to an image, and associated cached files.

Delete the image if possible from the output location?

Image Service Plugin APIs

Metadata Storage

Provides internal metadata storage for the image rendering service, the default plugin could just be sqlite. CRUD on the images object.

Image Object: id, payload

Resource Fetching

Fetches a resource from a URL to a file-like cached object. HTTP by default, but could be expanded to git:// or hg://, internal SAN support, etc.

Resource Caching

Takes IDs and returns file-like objects for reading and writing. Also needs interface for cache size management.

Image Building

Takes file-like object as input resource, a dict/JSON definition of the image, and writes it to a file-like output file.

Image Output

Provides a file-like output object, and the URL to be referenced by it upon completion. This is what each provider will have to implement in order to support the image format.

Required Provider APIs

Create Image

Takes the url of the manifest for the image. The manifest is validated by the API, then stored. Returns provider unique image id.

Create Node

Implemented by the provider, takes the ID that was returned by create image.

Image schema

Args takes a list of arg objects. An arg object includes, name, help text, value, default – used to describe a UI. Values list of key value fields for the args list. Everything that is ran through the templating engine is part of the args list.

Outstanding questions

- Windows - will the format work with Windows? How to handle licensing issues ** The rendering service will not get in the way of a provider supporting this
- Commercial distributions - should the format be able to support allowing image owners to mark up the cost of their image? How does the service provider support this? And how does the image owner get paid? ** Again, it will not get in the way of the provider supporting this