Development 101

This is the intro page to developing for CloudStack. If you're looking for other topics on CloudStack, look a few pages further down on the site.

Design Goals

To start off with CloudStack development, you must first understand the design goals of CloudStack. And, yes, we did have some design goals from the beginning. We didn't plan on being a "five geeks in a garage operation" for very long!

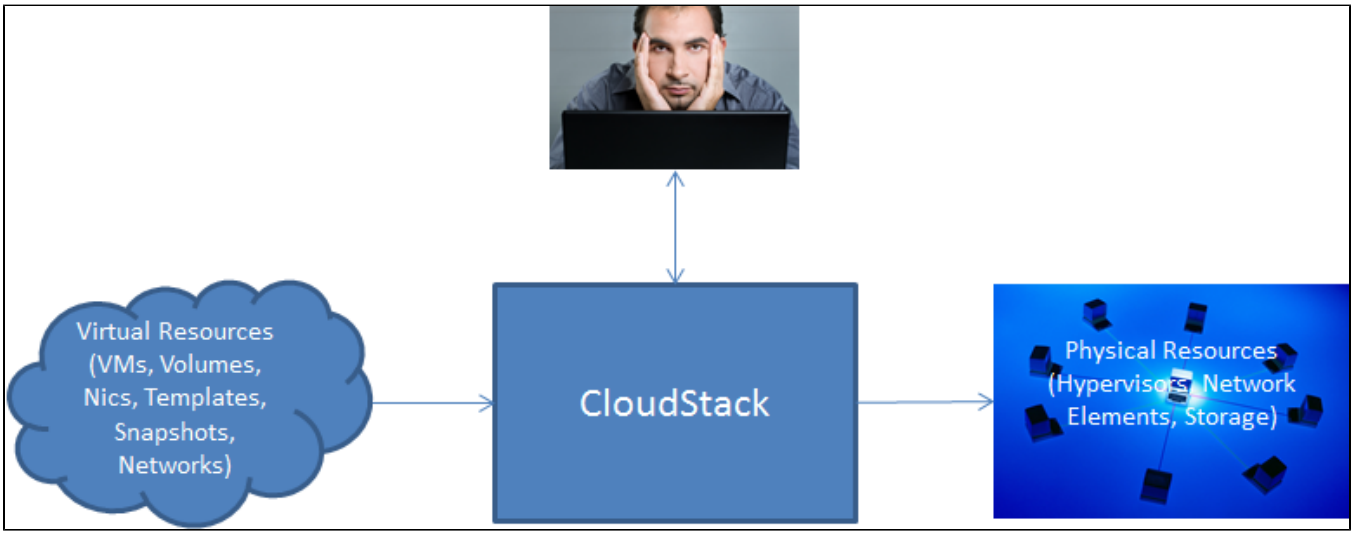
- Integrate with untold number of yet-to-be identified hardware.
- Provide an API platform on which to run cloud operations.
- Orchestrate hardware resources that may be protected by a firewall.
- Horizontally scalable management layer
- Enable the best data paths to accomplish cloud operations.
- A beautiful and functional UI

Of course, there's the usual "Don't be evil", "Conquer the world", "A cloud in every data center" dribble, but that's not quite useful for you. Here, on 101, we will start with how CloudStack implemented the first three design goals.

To do this, CloudStack has three different sets of APIs: the Platform API, the Agent API and the Plugin API. The Platform API is the REST-like API through which end users and administrators control CloudStack. There are various client-side bindings to this API that allow the caller to quickly put together scripts to automate processes within CloudStack. However, this API must be very secure as the caller is not trusted. The Agent API is used for CloudStack components to talk to the ServerResource, which then translates this API to what the hardware resource understands. This API is JSON based so that the client can be written in any language and to run on any platform convenient to the developer. The Plugin API allows one to insert code directly into CloudStack deployments to add to or modify behavior of CloudStack. This is a Java API and there is a set of predefined functionality exposed through this API.

Integrating new hardware resources with Agent API

At its core, as the diagram below shows, CloudStack provides the capability to map virtual resources to various physical resources. Note that it is only the "capability" and not the actual "mapping". This is key to the design principles of CloudStack. The system administrator provides, controls, and monitors the actual mapping. CloudStack is only the tool. Use it for good or for evil. It's all up to that guy in the middle there.



Every cloud operator has hardware they're comfortable with from their experiences in running data center operations. Some, like XenServer, NetScaler, and, ahem, everything else Citrix makes, are universally loved; however, more often than not, the preferences vary. Handling various hardware has its difficulties:

- Different interfaces. Some uses XML/RPC; some SOAP; some CLI; some C. You should be happy if it has a programming interface at all.
- Different languages. Not all can be programmed using Java.
- Different ways and perhaps even no way to cancel operations or time out.
- Different response times to a command.
- Most do not support any transactional semantics.

To handle these types of problems, CloudStack chose to provide the following implementation designs:

- CloudStack is composed of two parts: the management server that contains the business logic and the ServerResource that contains the translation layer to talk to the hardware.
- ServerResource can be deployed within the management server or remotely in an agent container. Therefore, a ServerResource cannot access the management database.
- ServerResource is a peer (not a slave) to the management server in terms of the hardware resources it communicates with. There is a scalability implication that we will get to in advanced topics.
- JSON is the API glue between ManagementServer and ServerResource. Therefore, a ServerResource does not have to be written in Java.
- An agent container is written to deploy ServerResources and handle protocol issues such as communicating through the firewall, serializing and deserializing, security, etc. The ServerResource does not need to worry about how to take care of those things. The agent container will.

- ServerResource should perform its functions in an idempotent manner. Database transactions are not allowed to wrap any operations to be performed on hardware.

Once you read the above and you go read the code, you will have the following questions. And they all can be explained by one statement, "We didn't have the time and it needs work!"

- Why isn't any JSON published? Why are all of the agent API commands Java classes?
- Why are all ServerResources written in Java?
- Why are XenServer and VMware ServerResources deployed within the management server and never as a remote agent?
- Why is KVM only deployed as a remote agent but never within the management server?
- The agent container code is fairly scant in features. What about creating and destroying ServerResource, upgrading the code, etc. in an automatic manner? It is all a manual process right now.

Integrating with CloudStack using the Platform API

API is the window to CloudStack's soul. Since version 2.0, CloudStack has sported a flexible design for adding and modifying commands. Since version 3.0, CloudStack supports third-party vendors adding commands without modifying CloudStack's command list via the PluggableService interface. CloudStack's API designs are as follows:

- API is REST-like. Sheesh...think you probably know this one already but...hey...gotta state the obvious.
- REST API design must take into consideration the round-trip expense and overhead of a REST call. This, in general, means the REST API methods return much larger objects to minimize the overhead.
- There are three types of REST APIs: synchronous, asynchronous, and asynchronous-create.
    - Synchronous APIs generally only hit the DB of the management server.
    - Asynchronous APIs cause operations within the cloud system.
    - Asynchronous-Create APIs create new entities within the management server DB and then run operations against these new entities.

REST API is broken into two parts: the End-User API and OAM&P (Operations, Administration, Maintenance, and Provisioning) or, basically, the Admin API.

End-User Platform API

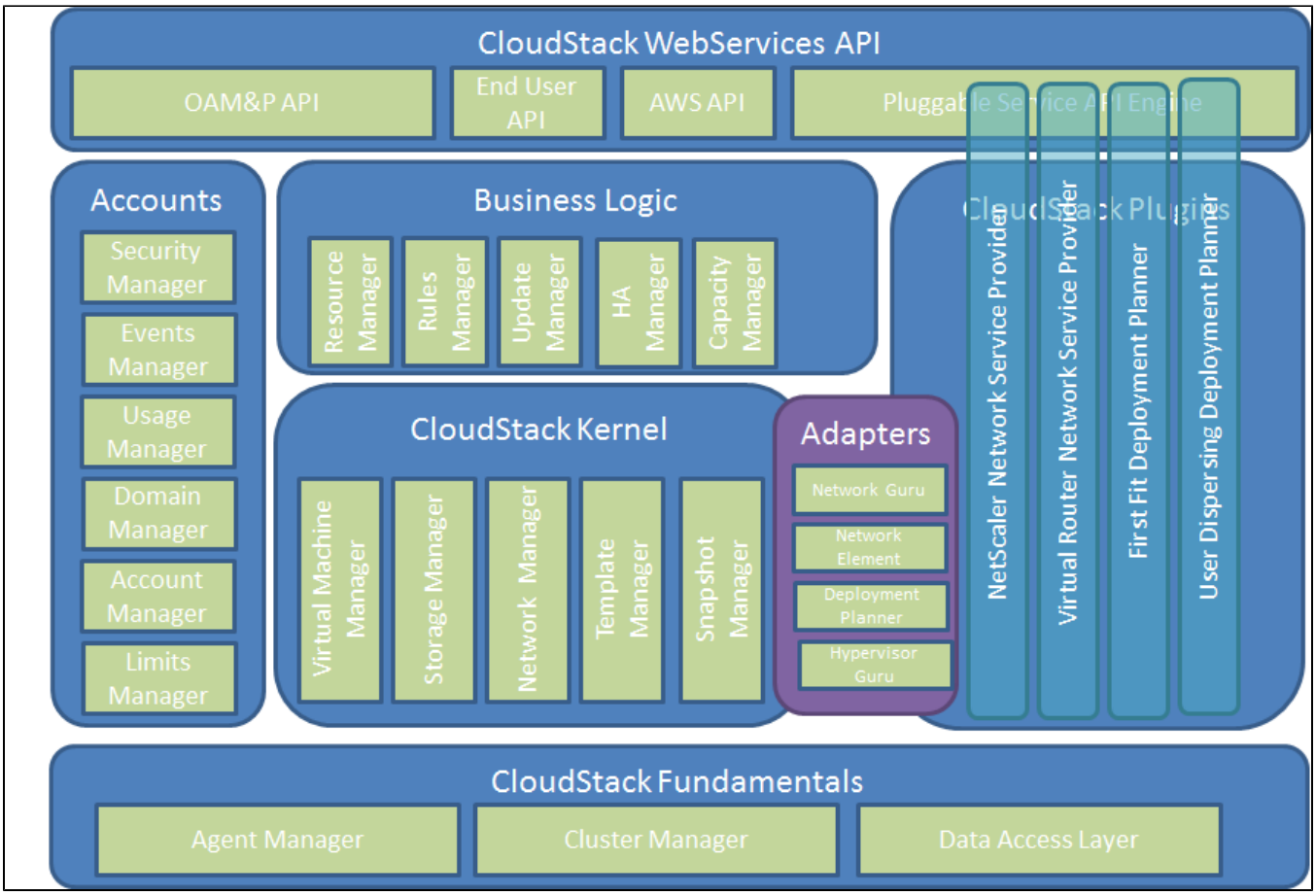
- The End-User REST API must always be backwards compatible.
- The End-User REST API must not leak any information about the physical resources underneath.

OAM&P Platform API

- The OAM&P REST API must attempt to maintain backwards compatibility. (WEAK....yes...Agreed)

Plugin API

Plugin API is where you can affect the most changes. The following is a picture of the CloudStack software architecture. Plugin APIs are defined by the Adapters, which expose the functionality required by CloudStack to implement cloud operations. The details of how to affect this change is more complicated and is explained in 201.



Getting to the Code!

Now that you have an idea of the design principles behind CloudStack, let's take a look at the rest of the nitty gritty you should take a look at before writing code.

Checking out the Source Tree

Understanding Packages and Build Dependencies

Setting up a CloudStack Developer environment for Windows, Mac or Linux

Fundamentals

Coding Conventions...(Snore but none the less important)

Exceptions & Logging

Data Access Layer

Locking (TODO)

Putting It Together

Putting CloudStack Together