An Implementor's Perspective on Interoperable Cloud APIs

Jorge L. Williams, Ph.D.
Senior Software Engineer
The Rackspace Cloud

Prepared for: Cloud Interoperability Roadmaps Session
December 10, 2009, Long Beach, CA

experience *fanatical support*®
WWW.RACKSPACE.COM

Talk Overview

- Open Standards and Rackspace
  - Our approach to Open Standards
  - How we see the development of Open Standards in the cloud
- The Cloud Servers API
  - A very quick glance at the API
  - Discuss some things missed on initial design
    - A process for deciding what features to support in the Cloud Servers API
    - Simplifying error handling for clients
    - The importance of the language binding
- Summary and Open Questions...

Open Standards and Rackspace

experience fanatical support®
WWW.RACKSPACE.COM

Open Standards and Rackspace

- Rackspace has never believed in “lock-in” as a model for success
  - Even in traditional hosting business...
    - Dedicated hosting model where there is only a single tenant
    - Most companies require a contract to customize infrastructure...
    - ...we always let you out if you felt we haven’t lived up to our promises
- Key to Success for Rackspace
  - Same as in traditional hosting business:

    Great Technology + Fanatical Support
  - The formula for success doesn’t simply rely upon a single factor, fanatical support plays an important role for us

Lock-in is a Common Objection to Adoption

- Proprietary formats, protocols, and APIs hinder acceptance and growth of the cloud
- Standardization adds value to the cloud as a whole
  - Not just for customers, but also to those of us providing cloud services

Development for Compute Standards in Three Phases

- Phase I: Migration
- Phase II: Federation
- Phase III: Burst

Phase I: Migration

- All About Portability of VMs
- Portable Virtual Machines give us the ability too...
  - Move VMs between different clouds (private, public)
  - Create VMs locally and import it
  - Easily share VMs with others
- Open Virtualization Format (OVF)
  - A big step in the right direction
  - ...but while OVF provides s standard packaging, it doesn’t address the problem of different virtual disk formats
- Many Efforts Concentrating on this Initial Phase
  - We envision the development of migration / conversion tools as an important first step

Phase II: Federation

- All About the Network
- Once we have portable VMs we want to move them to the cloud without reconfiguring anything (especially network settings)
  - Transparent Migration
  - We logically extend on premise network to the cloud
- We can do this ourselves to enable enterprise deployments
  - but standards make it possible to move across multiple clouds, multiple hypervisors, etc
- We envision latency as a problem
  - If, for example, enterprise and cloud deployments are geographically distant

Phase III: Burst

- All About APIs
- Once we have portable VMs and the ability to seamlessly integrate cloud deployments, we want to migrate and federate "on demand"
- Standardization of APIs means burst between (public, private) clouds
- Current standardization efforts in this area
  - For Data Storage, Cloud Data Management Interface (CDMI)
  - For Compute, Open Cloud Computing Interface (OCCI)
  - There are some obvious overlaps
    - CDMI adds functionality for account management, which can be useful for areas beyond a data management interface...
    - The idea though is to use them both in interoperable ways
- Our efforts
  - Cloud Files API for our Data Storage Service
  - Cloud Servers API for our Compute Service

The Cloud Servers API

experience fanatical support®
WWW.RACKSPACE.COM

The Cloud Servers API

- The Rackspace Clouds Compute Service API
  - Designed to be Open with the Goal of embracing standards when Possible
  - Standard Compute Features
    - Launch, delete, reboot, rebuild, and resize servers
    - Create custom “gold” images
    - Scheduled backups
    - Reset admin password
    - Customer sever metadata
    - Server data injection
    - Host identification
  - Noteworthy features
    - Multiple representations for entities (XML, JSON)
    - Shared IPs for high availability
    - Efficient polling
    - Pagination with limit and offset support
    - Leverage caching where appropriate
    - Limits and throttle controls (which can be queried) where appropriate for surge protection

The Cloud Servers API

- Design / Implementation
  - Iterative development process
    - Our partners were always in the loop and had access to the API at various stages of development
    - Additionally, we began consuming the service ourselves, mostly for testing

The Cloud Servers API

- The iterative design / development process lead to changes
  - Many minor changes from initial design spec to final implementation as we got continuos feedback from
    - Partners
    - Developers as they implemented the API and consumed it for testing
  - As we modified the spec we notice adjustments in focus
- We highlight some of the major changes in this talk:
  - The process of deciding what features to support was streamlined.
  - The process of handling errors and reporting errors to clients changed.
  - The need to focus on the language binding became apparent.
- Most changes seem obvious in retrospect, but we easily missed them in our initial design.
- An interesting thing to note is that our initial spec looked surprisingly similar to current standardization efforts.

What Features Should We Support?

experience fanatical support
WWW.RACKSPACE.COM

Underlying Philosophy: Lets Keep Things Simple

- To be successful, the API should be easily understood, and consumed...
- That ends up playing a role on the features and attributes of the API
- ...but we need to quantify “Simple”

Simple for Whom?

- The client consuming the service?
  - This means reducing barriers to entry for our consumers
    - Less lines of code to use the service
  - Problem: We have a diverse set of developers
    - Enterprise Developers vs Service Developers
    - Strong vs Weak Type Languages
    - Each group comes with a set of expectations, demands
    - Reducing lines in Java may mean increasing lines in Python or vice versa.
- The service implementor?
  - Reducing the amount of time to implement the service.
  - Service API specs are open sourced under Creative Commons 3.0, we want to encourage others to implement.
- The specification authors?
  - Not nearly as important as the above two.
  - The reality is that there is always pressure to get a spec out quickly.
  - No one is going to implement a spec that doesn't get out the door.
- Conflicts are likely, we respect the above order, it's a tougher call between different groups of clients.

Underlying Philosophy

- In order to be successful, the API must meet the needs of a diverse set of developers while presenting low barriers to entry.
- Client concerns get put at the top
- “Diverse Set of Developers”
  - Multiple types developers (Enterprise, Service, etc)
  - Multiple Languages
- “Low Barriers to Entry”
  - Reduce the lines of code as much as possible
  - Utilize existing software tools
  - Should seem “natural” to people utilizing the API

Should we Provide an XML Schema?

- We didn't initially.
- Big controversy about whether you need contracts in a ReST service, when we could utilize an existing format and HTTP provides a standard set of methods.
- Not interested in idealistic debate: Does providing a schema lower barriers for a group of developers?
- Yes. For, enterprise developers, particularly those using JEE and .Net, having a schema means writing significantly less code.
  - Schemas integrate well with existing software stacks...
  - ...so we supply them
  - Note, this doesn't get in the way of clients that don't want/need them.
- This complicates the life of the specification author
  - Designing a schema takes a fair amount of work
  - Lots of trial and error
  - But we swallow that complexity since it simplifies the life of both the client and service implementors

Error Handling

experience fanatical support
WWW.RACKSPACE.COM

Error Conditions are Often Overlooked

- Most ReST services take the approach of simply enumerating HTTP error codes for each Resource and Verb.
  - We did it in our initial spec
- Problem: Many error conditions can map to the same HTTP code
  - 400 BAD REQUEST
    - Malformed XML
    - Wellformed XML, but maybe a value of an enumeration is out of range
    - The service is expecting a “Server” entity, but you posted an “Image”
  - 409 CONFLICT
    - Trying to reboot a server that’s in the process of being created.
    - Trying to take a snapshot while a backup operation is taking place.
- With a web browser as your User Agent you can just quit on error, but that’s not always going to work.
  - What should you do if you receive an error while in the middle of provisioning a number of servers due to increase load?

Error Handling

- Ideally we want the service to
  - Provide the client with enough information to decide how to react to the error.
  - Provide a means to communicate the problem meaningfully to the user
    - The client may have a GUI front-end for example
- We could take two separate approaches
  - Provide a separate code for every discernible condition
    - Not very realistic
    - There are only so many error codes
    - Enumerating all conditions at the spec level is hard
  - Arrange errors by their basic type (Bad Request 400, Conflict 409, Unauthorized 401, etc) and add additional info in the content of the message.
    - The content holds a user friendly message...
    - ...and enough information to debug / track down problems
    - Rackspace does this, Amazon does this, but haven't seen this in OCCI, CDMI, etc.

Error Handling

- Obviously there's a need for standardization

Error Content	
Rackspace	Amazon

`<cloudServersFault code="500">`
`  <message>Fault!</message>`
`  <details>`
`      Error Details...`
`   </details>`
`</cloudServersFault>`
	`<Error>`
	`  <Code>InternalError</Code>`
	`  <Message>Fault!</Message>`
	`  <RequestId>`
	`    4442587FBFD0F2F9`
	`  </RequestId>`
	`</Error>`

Error Handling

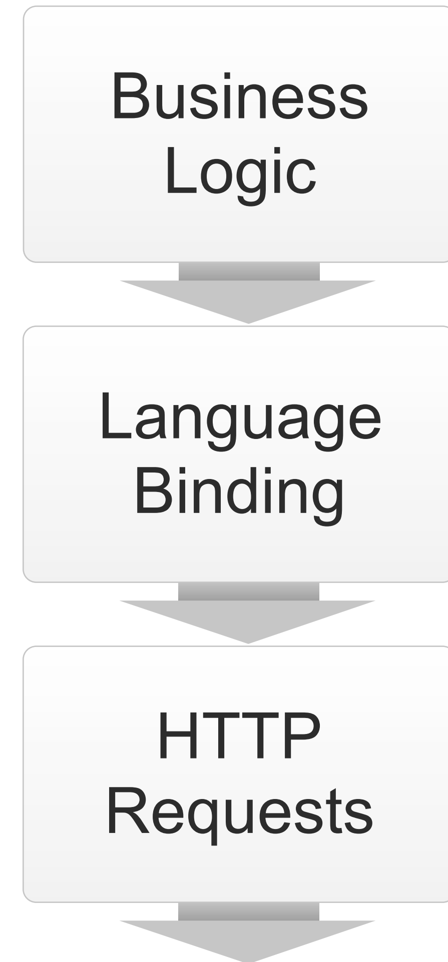
- Having a Standard Error Representation means these can be easily turned into exceptions in the target language...
- ...This is helpful in both the client and the service end

The Language Binding

experience fanatical support
WWW.RACKSPACE.COM

The Language Binding

- Most developers access our API through some sort of language binding.
- For each and every request a client should be able to...
  - ...handle authentication requests (if a token expires)
  - ...handle Overlimit (slow down) requests, especially when polling
  - ...handle communication failures of any kind, since we are dealing with a remote system
- The ReST API is a protocol in every respect, it makes sense to handle the protocol at a lower layer and thus reuse code effectively.
- Use the features of the target language is a good thing
  - Objects, rather than representation documents
  - Errors at compile time for strongly typed languages
  - Exceptions, rather than writing complicated "if" or "switch" statements on error codes.



Exposing Advance Features

- The binding provides a means to expose advance features without sacrificing the benefits of offering a service via HTTP.
  - HTTP scales because it is cacheable and stateless
  - Unfortunately this means calls to a compute service are asynchronous and require polling
- Most clients would rather be notified of changes rather than poll
  - We could handle non-HTTP “push” protocols to notify changes (XMPP)
    - Need to maintain state which may affect scalability
    - May have to deal with issues related to firewall etc.
  - A simple alternative is to simply hide polling operations in the language binding to support wait, and notify calls
    - Best of both worlds, coarse messages at the server end allow for greater scalability
    - State is maintained on the client end to allow for greater usability
    - This is the same approach utilized in Ajax apps

Do We Need a Standard Binding Spec?

- The notion that if we standardize the ReST API we'll automatically get standard bindings is false
  - Bindings to our API are submitted to us all the time
    - Different levels of abstraction, some close to HTTP others very divorced from it
    - Different object representations for the same entities
    - Different features of the protocol supported (efficient polling, paginated results, etc)
    - Often bindings - even for the same language - will look radically different
  - Think about the benefits of having a standard API, DOM for example
    - Knowledge of the API can be taken from one programming language to another
      - JavaScript DOM on the client end, to setup a UI
      - Java DOM on the server end, to process XML
    - Different Implementations of the API with slightly different features can co-exist
      - Changing from one implementation to another involves little to no change in code
    - Can we support this without providing a binding standard? Are these features important?

Do We Need a Standard Binding Spec?

- Most Developers will view the Binding as the API.
- If a features is not exposed in a binding it will probably not be used
  - A concern for us, we want clients to utilize efficient polling operations, etc.
  - We want clients to respect rate limits.
- Cloud language bindings are emerging
  - What does it mean if we have standards on the server end, but client bindings vary widely?

Cloud APIs

Type	Client Binding	Mediating Service	Server API
Compute	Lib Cloud	Delta Cloud	OCCI
Storage	Simple Cloud		CDMI

Cross-Cloud Language Bindings

- Lib Cloud can provide a consistent view across clouds
  - Each provider has a different "driver"

current support

provider	list	reboot	create	destroy	images	sizes
EC2	yes	yes	yes	yes	yes	yes
EC2-EU	yes	yes	yes	yes	yes	yes
Slicehost	yes	yes	yes	yes	yes	yes
Rackspace	yes	yes	yes	yes	yes	yes
Linode	yes	yes	yes	yes	yes	yes
VPS.net	yes	yes	yes	yes	yes	yes
RimuHosting	yes	yes	yes	yes	yes	yes
GoGrid	no	no	no	no	no	no
flexiscale	no	no	no	no	no	no
Eucalyptus	no	no	no	no	no	no
vCloud	yes	yes	yes	yes	yes	yes
Hosting.com	no	no	no	no	no	no
Terremark	yes	yes	yes	yes	yes	yes
Voxel	no	no	no	no	no	no

Cloud Servers API Language Binding

- We provide a specification for our bindings
- Reference Implementations coming
  - Python, Ruby, Java
- Capture all features of the API consistently
- Lowers Barriers Dramatically for Clients

```
#
# Reboot the server, wait for the operation to complete...
#

serverManager.reboot(serverToReboot, rebootType.soft)
serverManager.wait(serverToReboot)
```

Summary and Open Questions

experience fanatical support®
WWW.RACKSPACE.COM

Some Lessons Learned from Cloud Servers API

- We assume Simplicity == High acceptance, but who's life are we simplifying?
  - The client implementor
  - The service implementor
  - The specification author
- Diverse set of developers need to be considered as we decide features and attributes of the API - conflicts may emerge
- Need to place emphasis on error handling - listing HTTP codes is not enough
- Most developers will code against a language binding
  - This should be considered when designing the API (hierarchical error types, for example)
  - Need to expose all features in the language binding
  - There may be a need to standardize bindings

Rackspace and Open Standards

- Rackspace has never believed in “lock-in” as a model for success
  - Great Technology + Fanatical Support
- Open Standards benefits service providers and clients
  - Lock-in is a common objection to adoption
  - Proprietary formats, protocols, and APIs hinder acceptance and growth of the cloud
- Efforts to Date
  - Cloud Servers API Spec
    - Released under Create Commons
  - Cloud Servers API Binding Spec
    - Also under Create Commons
    - Actively being developed in an open manner on git hub
  - All bindings (Cloud Servers, Cloud Files)
    - Developed openly on git hub under the MIT license
  - Active in DMTF cloud incubator and others
  - Want to support migration, federation, bursting and avoid lock-in

Some Open Questions...

- Most API efforts on Compute and Data Storage - where else does it make sense to place more emphasis in APIs? Account Management?

- If we are standardizing on an API how do we deal with semantic differences between providers.
  - "shared IPs" in Rackspace vs "elastic IPs" in Amazon, same basic functionality but the features work differently.
  - "security groups" vs "VLANs" vs other approaches, again functionality is essentially the same but there are differences.

- How do we continue to be open while allowing for innovation?
  - Obviously there is a need for Open Standards to be extensible
  - But if we allow for extensibility where do we draw the line? Where does extensibility get in the way of interoperability?

Thank You.

experience fanatical support®
WWW.RACKSPACE.COM