Cloud Servers™ Language Binding Guide

07/27/09

Binding v1.0 (Beta)

This document is intended for software developers interested in developing high level language bindings for the Cloud Servers Application Programming Interface (API) version 1.0.

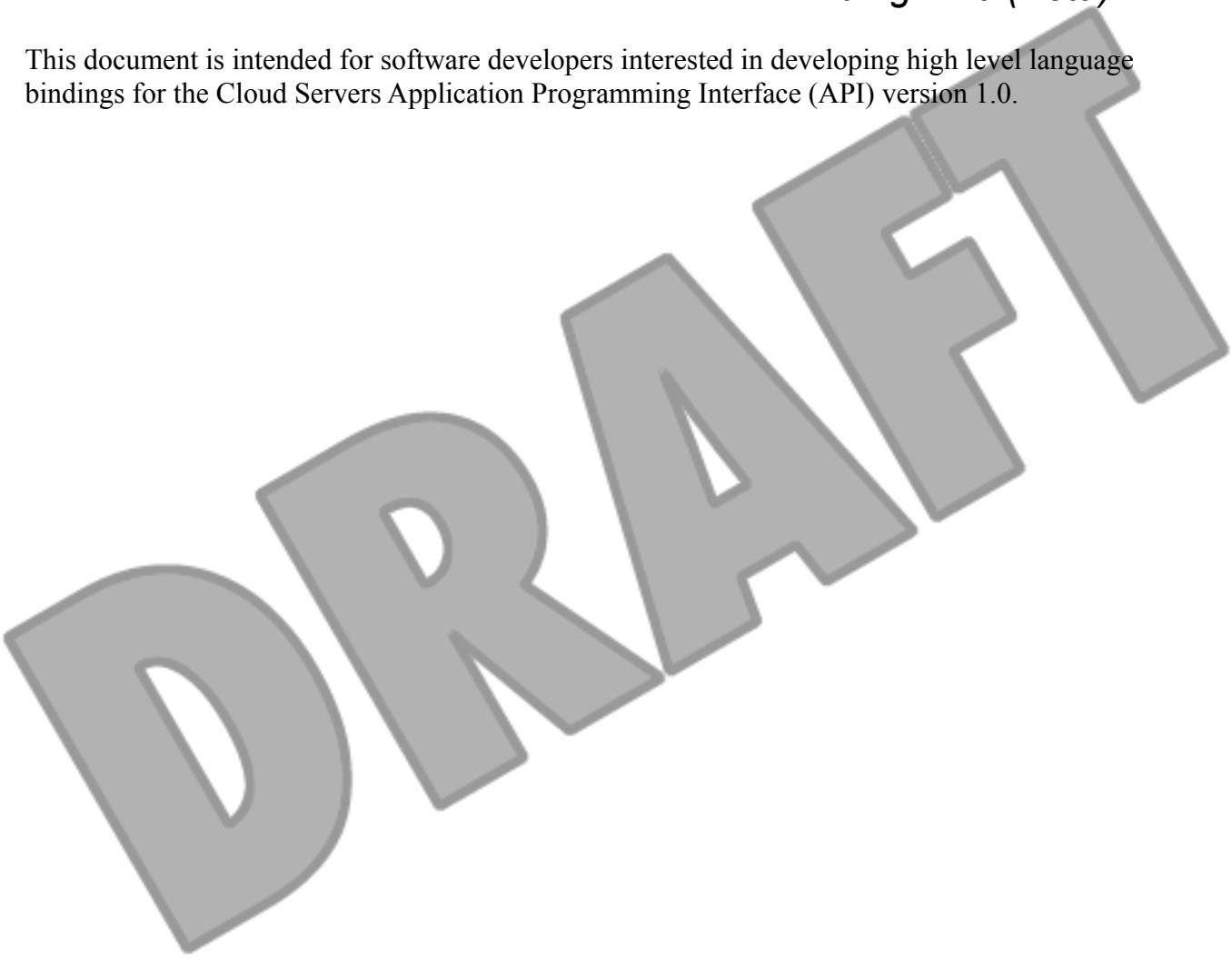

Table of Contents

Overview 1
Intended Audience 1
Language Binding Architecture 2
Interfaces 2
Entities 4
Faults 5
Entity Managers 6
CRUD Operations 8
Polling Operations 8
Notifications 10
Entity Lists 10
Entity List Iterations 11
Monitoring Entities with Delta Lists 12
The Cloud Servers Service 14
Authentication 15
Cloud Servers Entity Managers 16
The Server Manager 16
The Image Manager 17
The Shared IP Group Manager 17
The Flavor Manager 18

Overview

The Cloud Servers API is implemented as a RESTful service built directly on top of the HTTP 1.1 protocol. While some developers may feel comfortable interacting via this API, it is expected that many users will prefer, instead, to be shielded from the underlying protocol and to access Cloud Servers via a high-level language such as C++, Java, Python, or Ruby. The code that abstracts away the underlying protocol, and presents the entities and actions that compose the API in a language-specific manner is known as a *language binding*. This document provides guidance to developers implementing such language bindings for version 1.0 of the Cloud Servers API. The aim of this guide is to promote consistency among different implementations and to ensure that language bindings provide access to all features of the Cloud Servers API in a manner that is standardized and can be easily maintained as new versions of the API are released.

Language bindings that adhere to this specification will be considered *Rackspace Approved* bindings and will be featured on the Rackspace Portal.

Please note that this document is a **DRAFT**. It is intended to give customers and partners 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 at support@rackspacecloud.com.

Intended Audience

This guide is intended for software developers who want to develop language bindings for version 1.0 of the Cloud Servers API. It assumes the reader has a general understanding of the Cloud Servers service and is familiar with:

- RESTful web services
- HTTP/1.1
- JSON and/or XML data serialization formats
- The Cloud Servers API v1.0

Language Binding Architecture

The Language Binding Architecture provides a language and platform independent framework for interacting with Cloud Servers via the Cloud Servers API. The framework is designed to abstract away many of the low-level properties of the Cloud Servers API while allowing easy access to the full set of capabilities the API has to offer. The Cloud Servers API can be thought of as a low-level application protocol. The binding architecture provides a means of exposing the capabilities of this protocol to application developers in a manner that is consistent across programming languages and that offers transparent support for low level features of the API such as rate limits, efficient polling, and paginated collections. The architecture is designed to be flexible enough to accommodate new revisions of the API easily and may be reused as a basis for future API language bindings.

In this chapter, we describe the binding architecture in a language-neutral manner and provide guidance to developers who wish to provide an implementation of the architecture in a high-level programming language.

Interfaces

The language binding architecture is defined as a set of common abstract interfaces. These interfaces provide a standardized method of interacting with the Cloud Servers API. In order to remain language-neutral, we specify these interfaces in the Interface Description Language (IDL) notation defined by the Object Management Group (OMG). Language mapping specifications from IDL to many popular programming languages such as Java, Ruby, Python, and C++ are defined by the OMG and can be found in their document catalog:
http://www.omg.org/technology/documents/idl2x_spec_catalog.htm
Non-standard language mappings also exist for programming languages such as Perl and Visual Basic.

The interfaces specified in this guide as well as their related entities and faults must belong to the CloudServers API module defined below. Implementors may provide implementations of these abstract interfaces in their own modules.

CloudServers API Module

```
module com{
  module rackspace{
    module cloud{
      module servers{
        module api{
          module client{

            //
            //  Abstract interfaces, entities, and exceptions
            //  defined here...
            //

          };
        };
      };
    };
  };
};
```

Please note that IDL does not define a native timestamp type so we define timestamps in interfaces using **unsigned long long**. This is meant to represent a timestamp as Unix time (the number of seconds since January 1, 1970, 00:00:00 UTC, not counting leap seconds). Implementations need not represent timestamps in this manner and can instead use a native timestamp type if one is available.

Not all languages include support for method overloading or default parameters. Because of this, interfaces are defined with methods that have very similar names and vary only in the number of parameters they take.

Similar Wait Methods

```
interface EntityManager {
              // …
              void wait (in entity e);
              void waitT(in entity e, in unsigned long long timeout);
              // …
};
```

In the example above, languages that don't support overloading or default parameters should implement two methods (wait and waitT). Other languages may implement either two overloaded methods or a single method with an optional timeout parameter. In this document, we will always refer to these kind of calls by a single name – in this case “wait”.

Entities

An entity is a *thing* that is modeled in the API. For example, Servers and Images are entities in the Cloud Service API. The "source of truth" for all entities in the Cloud Servers API is specified in terms of Complex Types in the XML Schema Language as defined by the World Wide Web Consortium (W3C). Cloud Servers API schemas are found in the Rackspace Cloud documentation site: http://docs.rackspacecloud.com/servers/api/v1.0/xsd/. We reserve the use of the IDL type **entity** in interfaces to denote that a method call may accept any entity as defined by these schemas. Why not define entities in terms of IDL? While IDL may be capable of defining the overall structure of an entity, it has very limited support for defining restrictions on a particular type, and a very limited set of basic data types. XML Schemas are more flexible, they can define the overall structure of each entity (a Server can contain multiple Metadata items) as well as a set of validation rules that must be honored by the entity (no more than five metadata items are allowed, a server personality file name may not exceed 255 characters). XML Schemas also have a very large set of useful basic data types (base64 encoded strings, timestamps, durations).

Implementations need not adhere strictly to the constraints defined in the XML Schema, however, the names of types, enumerations, and fields must match those defined in the schema exactly. If enumerated types are supported in the target language they should be used when modeling enumerations, if not, enumerations should be handled according to OMG IDL Language Mapping Specifications. When no standard mappings exists, enumerations may be treated as strings. Note that the JSON request/response pairs described in the Cloud Servers Developer Guide adhere to all of these principles and can be used as a guide to those unfamiliar with XML Schema.

Entity objects may be extended to add fields beyond those defined in the XML Schema for the purpose of aiding the implementation or to adhere to the standards of the target platform. One example of this is the addition of a lastModified field in order to allow for conditional GET operations.

Entity objects may also be automatically generated from XML Schema on platforms that offer strong support for XML data binding. Because of this, entities should serve only as place holders for data. They should not perform actions and must not directly interact with the Cloud Servers API in any way. Interactions with the API should be performed solely by Entity Managers and Entity Lists both of which are described later in this chapter.

The mechanics of accessing the components (or the *fields*) of an entity may differ from language to language. For example, some languages may access entity fields directly others may access them via a method call. In IDL terms, we say that entities may be modeled either as **struct**s or as an **interface** with a collection of attributes.

Struct Entity Access

```
sampleEntity.anInt = 5;
print sampleEntity.anInt;
```


Interface Attribute Entity Access

```
sample.setInt(5);
print sample.getInt();
```

As a rule of thumb, entities field access should be performed in a manner that seams natural in the target language.

The XML simple type **base64Binary** should be implemented in a manner such that developers need not perform their own Base64 encoding and decoding. This type may be represented, for example, as an array of octets or as a binary stream.

It is recommended, though not required, that entities be validated against schema rules before they are transmitted in a request. A BadRequestFault should be raised if validation fails.

Faults

A fault (or exception) is an event that interrupts the normal flow of execution. Implementations should report all error conditions through faults. Developers utilizing a language binding should not be required to know, understand, or interpret HTTP status codes in order to handle error conditions. Faults should closely resemble fault responses as specified in the Cloud Servers Developer Guide though implementations are free to define additional faults. Entities derived from fault responses can be thrown directly or can be wrapped in appropriate exception objects.

Of particular importance is the fact that the CloudServersFault should serve as the base type for all faults in the Cloud Servers API including those faults that are implementation specific. Developers should be able to capture all Cloud Servers related faults by simply providing a *catch* clause for the CloudServersFault. On some implementations this may mean that all faults may be represented by a single fault type, in this case an implementation must supply a *faultType* filed which may contain the name of the fault. It is also important that the OverLimit fault maintain the *retryAfter* field in order to allow developers to easily manage their own polling operations given rate limits.

In the IDL example below, we define fault responses as objects and wrap them in individual exceptions as IDL does not allow for exception inheritance. Implementations are free to utilize direct exception inheritance if that is supported.

Faults

```
interface CloudServersAPIFault {
  readonly attribute string message;
  readonly attribute string details;
  readonly attribute unsigned long code;
};

interface OverLimitAPIFault : CloudServersAPIFault {
  readonly attribute unsigned long long retryAfter;
};

exception CloudServersFault {
  CloudServersAPIFault fault;
};

exception OverLimitFault {
  OverLimitAPIFault fault;
};
```

Entity Managers

Entity managers can be thought of as RPC interfaces to the root entities (Servers, Flavors, Images, and Shared IP Groups) of the Cloud Servers API. Each of these entities is associated with an entity manager responsible for managing the underlying HTTP protocol and for interacting with the Cloud Servers API in a consistent manner. All actions performed under a particular API root (/servers/) can be performed by an associated entity manager. For example, the Servers entity manager can perform all operations on servers, including changing a password, setting up a backup schedule, or sharing a public IP address.

Though entity managers may vary based on the kind of entity they manage, they share the common interface illustrated below. This interface defines the base functionality of an entity manager and provides a standardized way of encapsulating the HTTP operations in the REST service. Additionally, the interface provides access to advance features in the API such as efficient polling and access to paginated results. Note that not all calls may be supported by all entity managers (you are not allowed to delete a Flavor, for example) and that it is possible for entity managers to extend the base interface with additional calls.

Entity Manager

```
//
//  An Entity Manager:  All operations may raise a CloudServersFault,
//  this is left of the definition below for brevity.
//
interface EntityManager {

              //
              //  CRUD Operations
              //
              void create(inout entity e);
              void remove(in entity e);
              void update(in entity e);
              void refresh(inout entity e);
              entity find(in unsigned long id);

              //
              // Polling Operations
              //
              void wait (in entity e);
              void waitT(in entity e, in unsigned long long timeout);
              void notify (in entity e, in ChangeListener ch);
              void stopNotify (in entity e, in ChangeListener ch);

              //
              //  Lists
              //
              EntityList createList (in boolean detail);
              EntityList createDeltaList (in boolean detail,
                                          in unsigned long long
                                                      changes_since);


              EntityList createListP (in boolean detail,
                                      in unsigned long offset,
                                      in unsigned long limit);
              EntityList createDeltaListP (in boolean detail,
                                           in unsigned long long
                                                       changes_since,
                                           in unsigned long offset,
                                           in unsigned long limit);

};
```

Entity managers are created via the CloudServersService described below, and inherit authentication and other settings information from the service. Note that all entity manager calls are capable of throwing CloudServersFaults. The specific set of faults thrown by a call, however, depends on the type of entity manager and correspond closely to the set of faults described in the Cloud Servers Developer Guide. For example, the create call on a Server entity manager should be capable of throwing a ServerCapacityUnavailable fault.

CRUD Operations

An entity manager is responsible for managing the Create, Read, Update, and Delete (CRUD) operations associated with a particular root entity. These operations correspond with the POST, GET, PUT, and DELETE verbs in the HTTP protocol. Note that refresh and find provide two ways of performing a GET operation. The find operation is guaranteed never to throw an ItemNotFound fault, it should return null in the case where the entity is not located. The refresh operation updates an existing entity with the latest information available on the server and may throw an ItemNotFound exception if the entity is no longer available. In order to operate efficiently, the refresh operation always attempts to perform a conditional GET. Finally, note that the update operation is currently only available for the server entity manager as Servers are the only root resources that are updatable in this version of the Cloud Servers API.

Polling Operations

All CRUD operations described in the previous section are asynchronous and return immediately. Users are required to poll in order to determine the status of an ongoing operation. Note that polling frequency must be adjusted in order to respect the rate limits imposed on the Cloud Servers account. The following example illustrates polling in order to determine if a server has been created.

Polling a Server on Create

```
Server s = new Server();

s.name     = "Test Server";
s.imageId  = 2;
s.flavorId = 1;

try {
    serverManager.create(s);

    while (s.status == BUILD) {
         try {
                serverManager.refresh(s);
         }
         catch (OverLimitFault ovf){

           //
           // We may be hitting an absolute limit
           // in which case retryAfter is not set
           //
           if (ovf.retryAfter == NULL) {
             throw ovf;
           }

           //
           // We are hitting a rate limit...
           // Need to sleep in some platform dependent way or
           // perhaps do other work while we wait.
           //
           // The sleep_until call is not part of the API or
           // the language binding framework.
           //
           sleep_until (ovf.retryAfter);
         }
         catch (CloudServersFault csf)
         {
            throw csf;
         }
    }
}
catch (CloudServersFault csf) {
  //handle fault
}
```

Although the above polling operation may work well in cases where an application has additional work to do while it waits, it can be tedious in most use cases and, as in the example above, may require the use of a platform dependent sleep operation. To counter this problem, entity managers

provide an efficient means of performing polling via the wait call.

Waiting for a Server on Create

```
Server s = new Server();

s.name     = "Test Server";
s.imageId  = 2;
s.flavorId = 1;

try {
     serverManager.create(s);
     serverManager.wait(s);
}
catch (CloudServersFault csf) {
   //handle fault
}
```

From a developers perspective, a wait operation stalls until an end condition is met. Internally the wait call is in fact polling in a manner that respects the rate limits imposed on the Cloud Servers account. The type of entity determines what an end condition is. For Servers and Images, an end condition is determined by an end state such as ACTIVE or ERROR and may also be determined by a progress setting of 100. The default end condition, used by entities besides Servers and Images, simply indicates that the entity has been modified – that a conditional GET operation returned new data.

Wait operations are capable of timing out and may throw a TimeOutFault(504) after a time out period expires. The time out period may be specified as an optional parameter in number of milliseconds. If not specified the time out period may be set to an implementation specific period that is long enough to allow most operations to complete.

Notifications

Notifications provide an event-driven way of monitoring Cloud Servers entities. An entity manager broadcasts notification events to interested ChangeListeners when there is a change in the entity. That is, a notification event will be broadcast anytime a conditional GET operation on an entity's URI would return new data. Internally, notification events are generated by an entity manager via polling operations which respect rate limits. When monitoring multiple entities, implementations may poll entity lists (/servers/) rather than making multiple calls to separate URIs (/servers/id, /servers/id2, /servers/id3, …).

An application indicates interest in receiving notifications by passing a ChangeListener to the notify call. This ChangeListener will continually receive events until either the application asks the notifications to stop via the stopNotify call or until the underlying polling operation for the entity encounters an unrecoverable fault.

A ChangeLister and its associated NotifyEvent are defined below.

Change Listener

```
struct NotifyEvent {
  boolean error;
  entity targetEntity;
  CloudServersAPIFault fault;
};

interface ChangeListener {
  void notify (in NotifyEvent ne);
};
```

Languages that support passing functions as parameters may define a ChangeListener as a function whose definition matches that of the notify method defined above.

In the even of an error, the error flag in the NotifyEvent will be set to TRUE and the offending exception will be returned in the fault field. In languages where exceptions are not allowed in structs a representation of the fault response will be placed instead. As stated above, an error event halts the notification process. An application is required to re-register an entity for notifications after such an event.

Entity Lists

Entity lists allow developers to interact with paginated collections and provide a simple means of monitoring a collection of entities for changes. An entity list references a collection of entities of a particular type. A delta list contains only a subset of entities that have been modified after a certain point in time. Additionally, entity lists may contain detailed entity objects, which are equivalent to those retrieved via the find operation, or simple entity objects in which only identifying fields (name and id) are set. All entity lists regardless of type (full or delta) or content (detailed or simple) implement the interface defined below. It is the responsibility of the application to keep track the type of entity list and its contents.

Entity List

```
interface EntityList {
  readonly attribute unsigned long long lastModified;
  boolean isEmpty();
  boolean hasNext();
  entity next();
  void reset();
  void delta();
};
```

Entity lists are created by an entity manager via the the createList and createDeltaList calls. The detailed flag in the list creation methods indicates whether the list should contain detailed or simple objects. The changes_since parameter is used to indicate the point in time after which changed items should appear in a delta list. If offset and limit parameters are set, then a single page of

entities is returned. Requesting an offset beyond the last page of entities returns an empty list. The isEmpty call can be used to efficiently check if a list is empty. The limit parameter may not exceed the maximum page size as describe the Cloud Servers API – currently 1000. If offset and limit parameters are not specified, then a full list of entities is returned. These full lists abstract away the paginated nature of the entities collection and allow the iteration of all entities in an efficient manner.

Entity List Iterations

Entity lists are iterated in the following manner.

Entity List Iteration

```
while (list.hasNext()) {
   Server s = list.next();
   println "Found a server: "+s.name;
}
```

When iterating through full entity lists, implementations should walk through paginated collections dynamically. That is to say, they should retrieve a page of entities at a time and then immediately begin the iteration process. Through out the iteration, additional pages should be retrieved as needed. New pages should be retrieved in a manner that respects rate limits. Note that this iteration strategy has several important implications:

1. The iteration process may stall midstream as new pages are loaded.
2. The iteration process may fail after it has started.
3. If entities are added or removed midstream, the process may become unstable: some entities may not be visited, others visited multiple times.
4. The process starts quickly even when iterating through large entity sets.
5. The process is efficient in terms of memory as only a single page is kept in memory at a time.

Note that partial lists, those created with offset and limit parameters, do not perform automatic pagination. Partial and full list iterations, however, are equivalent in the case where the total number of entities is less than or equal to the iteration page size. The reset operation sets the list up so that it can be iterated through again, for a partial list this involves retrieving a new set of entities for the same page. Note that calling reset continually may raise an OverLimitFault. The following example continually displays a list of servers.

Listing all servers

```
EntityList l = serverManager.createList(FALSE);

while (TRUE) {
    try {
          while (l.hasNext())
          {
              Server s = l.next();
              println "Server: "+s.name;
          }
          l.reset();

    } catch (OverLimitFault olf) {
      // Throw absolute limit OR
      // Sleep OR perform work...
    }
}
```

Monitoring Entities with Delta Lists

Delta lists provide a means by which developers may monitor a collection of entities for changes. The following code, for example, monitors all servers for changes in an efficient manner, first by retrieving a full set of servers and then by continually requesting to list those servers that have changed since the last list was retrieved. Note that a lastModified call on a list can be used to retrieve a new delta list. If no servers have changed since the lastModified time an empty list is returned. Continually creating delta lists may raise an OverLimitFault.

Monitoring Server Names, Status and Progress

```
EntityList l = serverManager.createList(TRUE);

while (TRUE) {
    try {
          while (l.hasNext())
          {
              Server s = l.next();

              print "Server: "+s.name+"  ";
              print "Status:  "+s.status+" ";
              println "Progress: "+s.progress;
          }

          l = serverManager.createDeltaList (TRUE, l.getLastModified());

    } catch (OverLimitFault olf) {
           //
           // We may be hitting an absolute limit
           // in which case retryAfter is not set
           //
           if (ovf.retryAfter == null) {
             throw ovf;
           }

           //
           // We are hitting a rate limit...
           // Need to sleep in some platform dependent way or
           // perhaps do other work while we wait.
           //
           // The sleep_until call is not part of the API or
           // the language binding framework.

           sleep_until (ovf.retryAfter);
       }

}
```

The delta call simplifies the monitoring process by allowing a method of efficient polling. The call blocks until at least one entity changes. It then converts the current list into a delta list containing the changed entities. Internally, the call polls in such a manner that respects the rate limits imposed on a Client Service. Note that the delta operation may stall indefinitely.

Monitoring Server Names, Status and Progress (with delta)

```
EntityList l = serverManager.createList(TRUE);

while (TRUE) {
      while (l.hasNext()) {

         Server s = l.next();

         print "Server: "+s.name+"  ";
         print "Status:  "+s.status+" ";
         println "Progress: "+s.progress;
      }
      l.delta();
}
```

The Cloud Servers Service

The CloudServersService provides a means by which entity managers for the Cloud Servers API may be created. The interface for the service is defined below:

The Cloud Servers Service

```
interface CloudServersService
 {
   readonly attribute ServiceInfo serviceInfo;
   ServerManager createServerManager();
   ImageManager  createImageManager();
   SharedIpGroupManager createSharedIpGroupManager();
   FlavorManager createFlavorManager();
 };
```

How the CloudServersService is create is implementation specific. It may be created directly or via the use of a factory. A Username and API Key must be specified during creation as well as an optional collection of Settings.

Cloud Servers Service Settings

```
interface Settings {
  void setSetting (in string name, in string setting);
  string getSetting (in string name);
};
```


Settings are name/value pairs that provide a way of customizing service behavior and of specifying additional configuration options, for example proxy information. Regardless of how the service is created Settings should always be optional – that is passing a value of NULL instead of Settings object should never raise a fault.

Creating The Cloud Servers Service

```
CloudServersService s = new CloudServersService (“jdoe”,
                                                 “a86850deb2742”,
                                                  /* Settings */ NULL);
```

Upon creation of the cloud servers service, an implementation is encouraged to retrieve API version information and to log a warning if the status of version v1.0 is marked as DEPRECATED.

Besides providing a means of creating entity managers, the CloudServersService also allows developers to query meta information about the service via the serviceInfo attribute.

Service Info

```
interface ServiceInfo {
  readonly attribute Version versionInfo;
  readonly attribute Limits  limits;
  readonly attribute Settings settings;
};
```

Here, Version and Limits are the entities defined by the Cloud Servers API for specifying detailed API version info and obtaining absolute and rate limits. Version information may be cached. Limits, on the other hand, are always retrieved dynamically – note this means they may raise an OverLimitFault. The settings passed to the CloudServersService are also made available via the ServiceInfo interface. Note, that an implementation may filter these settings to remove sensitive information.

Authentication

Implementations must abstract away the authentication process. Thus, the process of accessing the authentication service in order to obtain auth tokens and to renew tokens once they expire should be completely hidden from developers. UnauthorizedFaults should not be raised unless multiple attempts to authenticate fail.

The X-Server-Management-Url header returned from the Rackspace Cloud Authentication Service should not be used directly by implementations. Instead, the account ID should be appended to the URL https://servers.api.rackspacecloud.com/v1.0/. For example if the authentication service returns the header:


```
X-Server-Management-Url: http://servers.api.rackspacecloud.com/v1.1/12345
```

An implementation should use http://servers.api.rackspacecloud.com/**v1.0**/12345 as its document root. This ensures that implementation instances continue to function correctly for as long as an API version is supported – even as new versions of the Cloud Servers API are released.

Cloud Servers Entity Managers

The Server Manager

The ServerManager inherits and implements all operations from the EntityManager interface. It also extends the interface to add support for server operations. All operations are asynchronous. Faults are not specified in the interface below for the sake of brevity, consult the Cloud Servers Developer Guide for a list of possible faults.

The following are considered end states by the wait call: ACTIVE, SUSPENDED, VERIFY_RESIZE, DELETED, ERROR, and UNKNOWN. Note that VERIFY_RESIZE can also serve as a start state, implementations are responsible for keeping track of whether this state should be treated as a start or end condition.

The Server Manager

```
interface ServerManager : EntityManager {

  void reboot  (in Server s, in RebootType type);
  void rebuild (in Server s, in unsigned long imageId);
  void resize  (in Server s, in unsigned long flavorId);
  void confirmResize(in Server s);
  void revertResize(in Server s);
  void shareIp (in Server s,
                in string ip,
                in unsigned long sharedIpGroupId,
                in boolean configureServer);
  void unshareIp (in Server s, in string ip);
  void setSchedule(in Server s, in BackupSchedule b);
  BackupSchedule getSchedule(in Server s);

};
```


The Image Manager

The image manager inherits all operations from the EntityManager interface, though an update call will raise a BadMethodFault because updating images is not supported in version v1.0 of the Cloud Servers API.

The following are considered end states by the wait call: ACTIVE, FAILED, and UNKNOWN.

The Image Manager

```
interface ImageManager : EntityManager {

  //
  //  update will raise a BadMethodFault
  //

};
```

The Shared IP Group Manager

The shared IP group manager inherits all operations from the EntityManager interface, though an update call will raise a BadMethodFault because updating IP groups is not supported in version v1.0 of the Cloud Servers API. A wait call stalls until a shared IP group is modified.

The Shared IP Group Manager

```
interface SharedIpGroupManager : EntityManager {

  //
  //  update will raise a BadMethodFault
  //

};
```


The Flavor Manager

Flavors are immutable, so create, remove and update all raise BadMethodFaults. A wait call stalls until a flavor is modified.

The Flavor Manager

```
interface FlavorManager : EntityManager {

  //  create, remove and update
  //  will raise BadMethodFaults

};
```


