



Pexip Infinity

Management API

Software Version 16

Document Version 16.a

August 2017

Contents

Introduction	5
About Pexip Infinity.....	5
About this guide.....	5
Definitions.....	5
Intended audience and references.....	6
Using the management API	7
Obtaining complete API information (schemas).....	7
Authentication.....	7
Examples used in the guide.....	7
Configuration replication delay.....	7
API performance.....	8
API design guidelines.....	8
Security.....	8
Configuration API	9
Configuration resources.....	9
Resource details.....	11
Resource methods.....	11
Getting resource details.....	11
Creating a single resource object.....	12
Updating a single resource object.....	12
Updating multiple resource objects.....	12
Deleting all resource objects.....	12
Examples.....	13
Creating a Virtual Meeting Room, Virtual Auditorium, Virtual Reception, or Test Call Service.....	13
Getting a Virtual Meeting Room's configuration.....	14
Changing an existing Virtual Meeting Room.....	14
Adding a Virtual Meeting Room alias.....	15
Deleting a Virtual Meeting Room.....	15
Creating a Virtual Meeting Room with aliases.....	15
Getting all Virtual Meeting Rooms, Virtual Auditoriums and Virtual Receptions.....	16
Getting all Virtual Meeting Rooms only.....	16
Creating multiple Virtual Meeting Rooms.....	16
Creating a Virtual Auditorium.....	16
Creating a Virtual Reception.....	17
Creating Automatically Dialed Participants.....	17

Modifying an Automatically Dialed Participant	17
Removing an Automatically Dialed Participant from a Virtual Meeting Room	18
Deleting an Automatically Dialed Participant	18
Creating Call Routing Rules	18
Deleting a Call Routing Rule	19
Deploying a new Conferencing Node	19
Downloading a Conferencing Node for manual deployment	20
Deploying a Conferencing Node using a VM template and configuration file	21
Uploading a service theme	22
Changing the distributed database setting of a Conferencing Node	22
Returning a license	22
Adding a license entitlement key	23
Status API	24
Status resources	24
Specifying the object ID in the path	24
Resource details	25
Resource methods	25
Pagination and filtering	25
Examples	25
Getting all active conference instances	25
Getting all active Virtual Meeting Room conferences	26
Getting all participants for a conference	27
Getting the media statistics for a participant	29
Getting the status of a Conferencing Node	30
Getting the load for a system location	30
Getting all registered aliases	30
Listing all cloud overflow Conferencing Nodes	30
Listing all locations monitored for dynamic bursting	31
Listing all locations containing dynamic bursting Conferencing Nodes	32
History API	33
History resources	33
Resource details	33
Resource methods	33
Pagination and filtering	34
Examples	34
Getting all conference instances	34
Getting all participants for a conference instance	34
Getting all participants for a time period	37
Getting all participants with packet loss	37

Command API	38
Command resources	38
Resource details	38
Resource methods	39
Response format	39
Examples	40
Dialing a participant into a conference	40
Disconnecting a participant	40
Muting a participant	41
Muting all Guest participants	41
Unmuting a participant	41
Unmuting all Guest participants	41
Locking a conference instance	42
Unlocking a conference instance	42
Unlocking a participant	42
Transferring a participant	42
Creating a system backup	43
Restoring a system backup	43
Starting an overflow Conferencing Node	44
Retrieving, paginating, filtering and ordering resource details	45
Getting a single resource object	45
Getting multiple resource objects	45
Pagination	45
Filtering	46
Ordering	47
Using the API with SNMP	48
Examples	48
Retrieving the SNMP sysName	48
Retrieving CPU load average	48

Introduction

About Pexip Infinity

Pexip Infinity is a virtualized and distributed multipoint conferencing platform. It enables scaling of video, voice and data collaboration across organizations, enabling everyone to engage in high definition video, web, and audio conferencing. It can be deployed in an organization's datacenter, or in a private or public cloud such as Microsoft Azure, Amazon Web Services (AWS) or Google Cloud Platform (GCP), as well as in any hybrid combination.

It provides any number of users with their own personal Virtual Meeting Rooms, as well as Virtual Auditoriums, which they can use to hold conferences, share presentations, and chat. Participants can join over audio or video from any location using virtually any type of communications tool (such as Microsoft Lync / Skype for Business, a traditional conferencing endpoint, a mobile telephone, or a Pexip Infinity Connect client) for a seamless meeting experience.

About this guide

Pexip Infinity includes a management API that allows third parties to control, configure, and obtain status information on the Pexip Infinity platform. Typical tasks include:

- dynamic creation of Virtual Meeting Rooms
- controlling outbound calls to endpoints and recording solutions
- controlling conference participants
- deploying new Conferencing Nodes
- real-time and historic status monitoring

This guide describes how to use the Pexip Infinity version 16 API. It comprises the following sections:

Using the management API	An overview of the RESTful web API used to communicate with a Pexip Infinity Management Node.
Configuration API	How to use the API to configure the Pexip Infinity platform and services (Virtual Meeting Rooms, Virtual Auditoriums, Virtual Receptions and the Pexip Distributed Gateway).
Status API	How to use the API to obtain status information on the Pexip Infinity platform and services.
History API	How to use the API to obtain historical information on the Pexip Infinity platform and services.
Command API	How to use the API to control aspects of Pexip Infinity conference instances.

Definitions

In the context of this API guide:

- A **service** is a Virtual Meeting Room, Virtual Auditorium, Virtual Reception or Pexip Distributed Gateway.
- A **conference instance** is a unique conference that is created when one or more participants accesses a **service**, and exists only until the last participant leaves.

Intended audience and references

It is assumed that readers are familiar with the concepts of HTTP(S), JSON and REST. The following links provide more information about the technologies used in relation to the management API.

HTTP(S)

- http://en.wikipedia.org/wiki/HTTP_Secure
- <http://tools.ietf.org/html/rfc2616>
- <http://tools.ietf.org/html/rfc2818>

JSON

- <http://en.wikipedia.org/wiki/JSON>
- <http://www.json.org/>
- <http://tools.ietf.org/html/rfc4627>

REST

- http://en.wikipedia.org/wiki/Representational_state_transfer
- <http://www.ibm.com/developerworks/webservices/library/ws-restful/>

Python

- [http://en.wikipedia.org/wiki/Python_\(programming_language\)](http://en.wikipedia.org/wiki/Python_(programming_language))
- <http://www.python.org/about/gettingstarted/>
- <http://docs.python-requests.org/en/latest/>
- <http://pysnmp.sourceforge.net>

Using the management API

This section describes how to use the Pexip Infinity management API.

Obtaining complete API information (schemas)

A summary of the schemas is available from within the Management Node via <https://<manageraddress>/api/admin/schema/>. If you require the schemas but do not have access to a Management Node running v16, please contact your authorized Pexip representative.

Full information for each configuration, command and resource is available from within the Management Node by downloading the resource's schema. The information in each schema includes, where applicable:

- all the available fields
- whether each field is optional or required
- the type of data each field must contain
- any restrictions on the minimum or maximum length of each field
- help text with information on usage

See the [Command API](#), [Configuration API](#) and [Status API](#) sections for a list of all resources and how to access the schemas for each.

Authentication

All access to the management API is authenticated over HTTPS.

If you are not using an LDAP database for authentication, access is via the credentials for the web admin user. The default username for this account is **admin**.

If you are using an LDAP database, we recommend you create an account specifically for use by the API.

Examples used in the guide

This guide uses Python 2.7 and the requests library (<https://pypi.python.org/pypi/requests>) for examples. However, many other modern programming languages and libraries exist which could be used equally successfully for driving the Pexip Infinity RESTful APIs.

The examples used are self-contained programs rather than pseudo code. These are given in an effort to help illustrate programming possibilities.

The following example data is used in the examples:

<manageraddress>: the IP address or FQDN of the management node.

<password1>: the admin web password for accessing the management node.

<user1>: the admin web username for accessing the management node. On a default installation this will be **admin**.

Configuration replication delay

It may take some time between configuration being provided by the REST API and this configuration taking effect across the Conferencing Nodes. This delay is not usually more than approximately 60s, depending on load and network conditions.

API performance

Accessing the REST API on the Management Node requires that the Management Node lock access to its database to ensure that configuration remains consistent and coherent. This therefore limits the rate at which requests can be serviced.

When configuration is modified, the modified configuration is replicated to the various Conferencing Nodes. This means that configuration changes will cause some CPU usage on both the Management Node and the Conferencing Nodes.

Heavy or frequent API usage will consume resources across the Pexip Infinity deployment.

API design guidelines

Some guidelines for designing your application:

- Avoid duplication. Avoid designs which will perform large amounts of polling for information that is not needed or which will result in repeatedly writing the same configuration values.
- Consider application designs which avoid the need to make large numbers of requests concurrently. Although concurrent API requests will work they may not be processed in the order that they commence. Consider waiting for a request to complete before making another one.
- In cases where multiple REST requests will be made sequentially in quick succession, consider using HTTP 1.1 connection keep alives and reusing your HTTPS connection in order to avoid the not insignificant handshake overhead of re-establishing separate HTTPS connections for every request.
- Be aware that other systems and applications in your deployment could also require access to the Management API, so consume the API as little as possible to ensure that they can also make use of it.
- In situations where the Pexip Infinity is heavily loaded, API performance may be impaired.
- Very heavy usage of the API might cause the web-based Administrator interface performance to be impeded. Conversely, heavy Administrator interface usage might impair API performance.

Security

We recommend that you upload a HTTPS certificate to your Management Node and each Conferencing Node and that client applications verify those certificates, in order to undertake server validation.

Configuration API

Configuration of the Pexip Infinity platform and services can be performed through the REST API, allowing you to create, view, edit and delete items.

Configuration resources

The following configuration resources are available via the REST API:

Component	Path
System configuration	
DNS server	/api/admin/configuration/v1/dns_server/
NTP server	/api/admin/configuration/v1/ntp_server/
Syslog server	/api/admin/configuration/v1/syslog_server/
SNMP NMS	/api/admin/configuration/v1/snmp_network_management_system
VM manager	/api/admin/configuration/v1/host_system/
Static route	/api/admin/configuration/v1/static_route/
Exchange servers [†]	/api/admin/configuration/v1/ms_exchange_connector/
Platform configuration	
System location	/api/admin/configuration/v1/system_location/
Management Node	/api/admin/configuration/v1/management_vm/
Conferencing Node	/api/admin/configuration/v1/worker_vm/
Licensing	/api/admin/configuration/v1/licence/
License request	/api/admin/configuration/v1/licence_request/
CA certificate	/api/admin/configuration/v1/ca_certificate/
TLS certificate	/api/admin/configuration/v1/tls_certificate/
Certificate signing request (CSR)	/api/admin/configuration/v1/certificate_signing_request/
Global settings	/api/admin/configuration/v1/global/
Call control	
H.323 Gatekeeper	/api/admin/configuration/v1/h323_gatekeeper/
SIP credentials	/api/admin/configuration/v1/sip_credential/
SIP proxy	/api/admin/configuration/v1/sip_proxy/
Microsoft SIP proxy	/api/admin/configuration/v1/mssip_proxy/
TURN server	/api/admin/configuration/v1/turn_server/
STUN server	/api/admin/configuration/v1/stun_server/
Policy server	/api/admin/configuration/v1/policy_server/
Service configuration	

Component	Path
Virtual Meeting Room, Virtual Auditorium, Virtual Reception, scheduled conference, or Test Call Service	/api/admin/configuration/v1/conference/
Alias for a service	/api/admin/configuration/v1/conference_alias/
Automatically Dialed Participant	/api/admin/configuration/v1/automatic_participant/
IVR theme	/api/admin/configuration/v1/ivr_theme/
Gateway routing rule	/api/admin/configuration/v1/gateway_routing_rule/
Registration settings	/api/admin/configuration/v1/registration/
Device	/api/admin/configuration/v1/device/
Conference sync template	/api/admin/configuration/v1/conference_sync_template/
Conference LDAP sync source	/api/admin/configuration/v1/ldap_sync_source/
Conference LDAP sync field	/api/admin/configuration/v1/ldap_sync_field/
Recurring conference †	/api/admin/configuration/v1/recurring_conference/
Scheduled conference †	/api/admin/configuration/v1/scheduled_conference/
Scheduled conference aliases †	/api/admin/configuration/v1/scheduled_alias/
Users	
Authentication	/api/admin/configuration/v1/authentication/
Account role	/api/admin/configuration/v1/role/
LDAP role	/api/admin/configuration/v1/ldap_role/
Permission	/api/admin/configuration/v1/permission/
Utilities	
Upgrade	/api/admin/configuration/v1/upgrade/
System backup	/api/admin/configuration/v1/system_backup/
† We do not currently recommend creation, deletion or modification of these resources directly; this should only be done by the VMR Scheduling for Exchange service.	

Resource details

More information can be obtained for each resource by downloading the resource schema in a browser. You may want to install a JSON viewer extension to your browser in order to view the JSON strings in a readable format.

For example, to view the schema for **aliases** the URI would be:

`https://<manageraddress>/api/admin/configuration/v1/conference_alias/schema/?format=json`

Each schema contains information on the available fields including:

- whether the field is optional (`nullable: true`) or required (`nullable: false`)
- the type of data the field must contain
- the choices for the field, e.g. `valid_choices: ["audio", "video", "video-only"]`
- which criteria can be used for [filtering](#) and [ordering](#) searches
- help text with additional information on usage.

Resource methods

Each configuration resource supports the following HTTP methods:

Method	Action
GET	Retrieves the current configuration for a resource
POST	Creates a new object for the resource
PATCH	Updates an existing resource object, or creates the object if it does not already exist
DELETE	Deletes an existing resource object

Getting resource details

See [Retrieving, paginating, filtering and ordering resource details](#) for information about how to retrieve the current configuration for a resource.

Creating a single resource object

To create a new object resource, a POST request is submitted to the root URI for the resource. The data should be a JSON object whose attributes match the field values for the resource.

Fields with default values may be omitted from the data.

The response to the POST method will contain a Location header which contains the REST URI of the newly created resource.

For example, the URI of a VMR resource takes the format: `https://<manageraddress>/api/admin/configuration/v1/conference/<object ID>/`

Note that Pexip Infinity always allocates a new object ID when creating a new resource.

Updating a single resource object

To update a single resource object, a PATCH request is submitted to the URI created by concatenating the object ID with the root URI of the resource.

For example, to make changes to the alias with ID **1** the URI would be:

`https://<manageraddress>/api/admin/configuration/v1/conference_alias/1/`

Updating multiple resource objects

The PATCH method can be used to create or update multiple objects at once. To update multiple objects a PATCH request is submitted to the root URI for the resource. The data must be formatted as a JSON object with a single attribute `objects` whose value is a list of the new objects.

Note: for any objects in the list that already exist, the `resource_uri` field must be included in the request. For any objects that are to be created by the PATCH request, the `resource_uri` field should be omitted.

Deleting all resource objects

If a DELETE operation is invoked on the root resource URI, then all the resource objects will be deleted. You should therefore use this operation with caution, and ensure you include the resource ID in the URI unless your intention actually is to delete all objects.

For example, to delete ALL Virtual Meeting Rooms, the URI is:

`"https://<manageraddress>/api/admin/configuration/v1/conference/"`

whereas to delete the Virtual Meeting Room with ID **1** only, the URI is:

`"https://<manageraddress>/api/admin/configuration/v1/conference/1/"`

Examples

Creating a Virtual Meeting Room, Virtual Auditorium, Virtual Reception, or Test Call Service

To create a new Virtual Meeting Room, Virtual Auditorium, Virtual Reception, or Test Call Service, you must submit a POST to the URI for this resource. You must specify the `service_type` as follows:

- `conference` for a Virtual Meeting Room
- `lecture` for a Virtual Auditorium
- `two_stage_dialing` for a Virtual Reception
- `test_call` for a Test Call Service

If you do not specify a `service_type`, the service will by default be created as a Virtual Meeting Room.

The following example creates a new Virtual Meeting Room with the name **VMR_1**. It has no aliases.

```
import json
import requests
response = requests.post(
    "https://<manageraddress>/api/admin/configuration/v1/conference/",
    auth=('<user1>', '<password1>'),
    verify=False,
    data=json.dumps({'name': 'VMR_1', 'service_type': 'conference'})
)
print "Created new Virtual Meeting Room:", response.headers['location']
```

Getting a Virtual Meeting Room's configuration

By submitting a GET to the resource URI of a Virtual Meeting Room, you can get its configuration:

```
import json
import requests
response = requests.get(
    "https://<manageraddress>/api/admin/configuration/v1/conference/1/",
    auth=('<user1>', '<password1>'),
    verify=False
)
print "Virtual Meeting Room:", json.loads(response.text)
```

Example output

```
Virtual Meeting Room: {
  'aliases': [
    {
      'alias': 'meet@example.com',
      'conference': '/api/admin/configuration/v1/conference/1/',
      'creation_time': '2017-05-11T22:02:05.855877',
      'description': '',
      'id': 1
    }
  ]
  'allow_guests': False,
  'automatic_participants': [],
  'call_type': 'video',
  'creation_time': '2017-05-11T22:02:05.848612',
  'description': '',
  'enable_overlay_text': false,
  'force_presenter_into_main': False,
  'guest_pin': '',
  'guest_view': None,
  'guests_can_present': true,
  'host_view': 'one_main_seven_pips',
  'id': 1,
  'ivr_theme': None,
  'match_string': '',
  'max_callrate_in': None,
  'max_callrate_out': None,
  'mssip_proxy': null,
  'mute_all_guests': false,
  'name': 'VMR_1',
  'participant_limit': None,
  'pin': '',
  'primary_owner_email_address': '',
  'replace_string': '',
  'resource_uri': '/api/admin/configuration/v1/conference/1/',
  'scheduled_conferences': [],
  'service_type': 'conference',
  'sync_tag': '',
  'system_location': null,
  'tag': ''
}
```

Changing an existing Virtual Meeting Room

Submitting a PATCH to an existing Virtual Meeting Room URI allows the data for that Virtual Meeting Room to be modified.

The following example updates the Virtual Meeting Room PIN to **1234**:

```
import json
import requests
response = requests.patch(
    "https://<manageraddress>/api/admin/configuration/v1/conference/1/",
    auth=('<user1>', '<password1>'),
    verify=False,
    data=json.dumps({'pin': '1234'})
)
```

Adding a Virtual Meeting Room alias

A Virtual Meeting Room must already exist before you can add an alias to it. To do this you submit a POST to the resource URI for Virtual Meeting Room aliases and add the partial URI of the Virtual Meeting Room to the data POSTed.

The following example creates a new alias **meet@example.com** for the Virtual Meeting Room with ID **1**.

```
import json
import requests
response = requests.post(
    "https://<manageraddress>/api/admin/configuration/v1/conference_alias/",
    auth=('<user1>', '<password1>'),
    verify=False,
    data=json.dumps({
        'alias': 'meet@example.com',
        'conference': '/api/admin/configuration/v1/conference/1/',
    })
)
print "Created new alias:", response.headers['location']
```

Deleting a Virtual Meeting Room

Deleting a Virtual Meeting Room is achieved by submitting a DELETE request to an existing Virtual Meeting Room URI.

The following example deletes the Virtual Meeting Room with ID **1**:

```
import requests
response = requests.delete(
    "https://<manageraddress>/api/admin/configuration/v1/conference/1/",
    auth=('<user1>', '<password1>'),
    verify=False
)
```

 Deleting a Virtual Meeting Room will also delete all aliases associated with it.

Creating a Virtual Meeting Room with aliases

To simplify the creation of a Virtual Meeting Room and the aliases that belong to it, it is possible to submit a single POST with all the information.

The following example creates a new Virtual Meeting Room with the name **VMR_1** and two aliases: **meet** and **meet@example.com**:

```
import json
import requests
response = requests.post(
    "https://<manageraddress>/api/admin/configuration/v1/conference/",
    auth=('<user1>', '<password1>'),
    verify=False,
    data=json.dumps({
        'name': 'VMR_1',
        'service_type': 'conference',
        'aliases': [{'alias': 'meet'}, {'alias': 'meet@example.com'}]
    })
)
print "Created new Virtual Meeting Room:", response.headers['location']
```

Getting all Virtual Meeting Rooms, Virtual Auditoriums and Virtual Receptions

Retrieving all the configured Virtual Meeting Rooms, Virtual Auditoriums and Virtual Receptions is achieved by submitting a GET request to the resource URI they all share:

```
import json
import requests
response = requests.get(
    "https://<manageraddress>/api/admin/configuration/v1/conference/",
    auth=('<user1>', '<password1>'),
    verify=False
)
print "Virtual Meeting Rooms:", json.loads(response.text)['objects']
```

Getting all Virtual Meeting Rooms only

To retrieve all the configured Virtual Meeting Rooms but not Virtual Auditoriums or Virtual Receptions, you submit a GET request to the resource URI as above but filter it by `service_type`:

```
import json
import requests
response = requests.get(
    "https://<manageraddress>/api/admin/configuration/v1/conference/?service_type=conference",
    auth=('<user1>', '<password1>'),
    verify=False
)
print "Virtual Meeting Rooms:", json.loads(response.text)['objects']
```

Creating multiple Virtual Meeting Rooms

Multiple Virtual Meeting Rooms can be created using a PATCH request.

The following example creates two Virtual Meeting Rooms; the first with the name **VMR_1** and an alias **meet1@example.com**, and the second with the name **VMR_2** and an alias **meet2@example.com**:

```
import json
import requests
data = { 'objects' : [
    { 'name' : 'VMR_1', 'service_type': 'conference', 'aliases' : [{'alias' : 'meet1@example.com'}]},
    { 'name' : 'VMR_2', 'service_type': 'conference', 'aliases' : [{'alias' : 'meet2@example.com'}]},
  ] }
response = requests.patch(
    "https://<manageraddress>/api/admin/configuration/v1/conference/",
    auth=('<user1>', '<password1>'),
    verify=False,
    data=json.dumps(data)
)
```

Creating a Virtual Auditorium

The following example creates a new Virtual Auditorium with the name **Lecture** and a single alias **lecture@example.com**:

```
import json
import requests
response = requests.post(
    "https://<manageraddress>/api/admin/configuration/v1/conference/",
    auth=('<user1>', '<password1>'),
    verify=False,
    data=json.dumps({
        'name': 'Lecture',
        'service_type': 'lecture',
        'aliases': [{'alias' : 'lecture@example.com'}]}
    ))
print "Created new Virtual Auditorium:", response.headers['location']
```


Creating a Virtual Reception

The following example creates a new Virtual Reception with the name **Reception** and a single alias **reception@example.com**:

```
import json
import requests
response = requests.post(
    "https://<manageraddress>/api/admin/configuration/v1/conference/",
    auth=('<user1>', '<password1>'),
    verify=False,
    data=json.dumps({
        'name': 'Reception',
        'service_type': 'two_stage_dialing',
        'aliases': [{ 'alias': 'reception@example.com' }]
    })
)
print "Created new Virtual Reception:", response.headers['location']
```

Creating Automatically Dialed Participants

You can create a new Automatically Dialed Participant by submitting a POST request to the resource URI for Automatically Dialed Participants. Automatically dialed participants are a little different from other resources as they may be associated with many different Virtual Meeting Rooms and Virtual Auditoriums.

The following example creates an Automatically Dialed Participant and associates them with an already created Virtual Meeting Room with ID 1:

```
import json
import requests
response = requests.post(
    "https://<manageraddress>/api/admin/configuration/v1/automatic_participant/",
    auth=('<user1>', '<password1>'),
    verify=False,
    data=json.dumps({
        'alias': 'myendpoint@mydomain.com',
        'remote_display_name': 'My Name',
        'description': "Dial myendpoint@mydomain.com whenever a related conference starts",
        'protocol': 'sip',
        'role': 'guest',
        'conference': ['/api/admin/configuration/v1/conference/1/']
    })
)
print "Created new automatically dialed participant:", response.headers['location']
```

Modifying an Automatically Dialed Participant

The following examples use PATCH to modify an existing Automatically Dialed Participant.

The example below associates the Automatically Dialed Participant (created above) with two (already created) Virtual Meeting Rooms — one with ID 1, one with ID 2:

```
import json
import requests
response = requests.patch(
    "https://<manageraddress>/api/admin/configuration/v1/automatic_participant/1/",
    auth=('<user1>', '<password1>'),
    verify=False,
    data=json.dumps({
        'conference': ['/api/admin/configuration/v1/conference/1/',
            '/api/admin/configuration/v1/conference/2/']
    })
)
print "added existing automatic participant to an additional meeting room:", response
```

Removing an Automatically Dialed Participant from a Virtual Meeting Room

The example below removes the association between the Automatically Dialed Participant and the Virtual Meeting Rooms:

```
import json
import requests
response = requests.patch(
    "https://<manageraddress>/api/admin/configuration/v1/automatic_participant/1/",
    auth=('<user1>', '<password1>'),
    verify=False,
    data=json.dumps({
        'conference' : []
    })
)
print "Removed automatic participant from all meeting rooms:", response
```

Deleting an Automatically Dialed Participant

The following example deletes the Automatically Dialed Participant with ID 1:

```
import requests
response = requests.delete(
    "https://<manageraddress>/api/admin/configuration/v1/automatic_participant/1/",
    auth=('<user1>', '<password1>'),
    verify=False
)
```

Creating Call Routing Rules

To create a new Call Routing Rule you submit a POST to the URI for that resource.

The following example creates a Call Routing Rule with the name **Route to Lync**, which routes incoming Pexip Distributed Gateway calls (via H.323, SIP and WebRTC) to Microsoft Lync / Skype for Business:

```
import json
import requests
response = requests.post(
    "https://<manageraddress>/api/admin/configuration/v1/gateway_routing_rule/",
    auth=('<user1>', '<password1>'),
    verify=False,
    data=json.dumps({
        'name': 'Route to Lync',
        'priority': 50,
        'match_incoming_calls': True,
        'match_outgoing_calls': False,
        'match_incoming_h323': True,
        'match_incoming_mssip': False,
        'match_incoming_sip': True,
        'match_incoming_webrtc': True,
        'match_string': '9(.*)',
        'replace_string': '',
        'called_device_type': 'external',
        'outgoing_protocol': 'mssip',
    })
)
print "Created new Call Routing Rule:", response.headers['location']
```

This example creates a Call Routing Rule that applies to outbound calls made from Pexip VMRs and routes them to registered devices only:

```
import json
import requests
response = requests.post(
    "https://<manageraddress>/api/admin/configuration/v1/gateway_routing_rule/",
    auth=('<user1>', '<password1>'),
    verify=False,
    data=json.dumps({
        'name': 'Outbound calls to registered devices',
        'priority': 60,
        'match_incoming_calls': False,
        'match_outgoing_calls': True,
        'match_string': '.*@mycompany\.com',
        'replace_string': '',
        'called_device_type': 'registration',
    })
)
print "Created new Call Routing Rule:", response.headers['location']
```

Deleting a Call Routing Rule

The following example deletes the Call Routing Rule with ID 1:

```
import requests
response = requests.delete(
    "https://<manageraddress>/api/admin/configuration/v1/gateway_routing_rule/1/",
    auth=('<user1>', '<password1>'),
    verify=False
)
```

Deploying a new Conferencing Node


By submitting a POST request to the resource URI for Conferencing Nodes, you can create and deploy a new Transcoding Conferencing Node onto a server running vSphere ESXi 4.1 or higher.

```
import json
import requests
response = requests.post(
    "https://<manageraddress>/api/admin/configuration/v1/worker_vm/",
    auth=('<user1>', '<password1>'),
    verify=False,
    data=json.dumps({
        'name': 'new_node',
        'hostname': 'newnode',
        'domain': 'example.test',
        'address': '<newnode_ip_address>',
        'netmask': '<newnode_ip_mask>',
        'gateway': '<ip_gateway_address>',
        'password': '<newnode_password>',
        'node_type': 'CONFERENCING',
        'system_location': '/api/admin/configuration/v1/system_location/1/',
        'deployment_type': 'VMWARE',
        'host': 1,
        'host_username': '<vcenter_username>',
        'host_password': '<vcenter_password>',
        'host_network': 'VM Network',
        'host_resource_path': '/<vmware_datacenter>/host/<vmware_host>/Resources',
        'vm_cpu_count': '8',
        'vm_system_memory': '16384',
    })
)
print "Created new Conferencing Node:", response.headers['location']
```

Note that the **transcoding** field (introduced in version 15) is deprecated in version 16; you must use the **node_type** field instead.

Downloading a Conferencing Node for manual deployment

By submitting a POST request with a `deployment_type` of `MANUAL`, you can download an OVA file which can be used to deploy a new Proxying Edge Node onto a host server running vSphere ESXi 5.0 or higher.

 Deployment types of `MANUAL-ESXI4` and `MANUAL-HYPERV2012` can also be used.

```
import json
import requests
response = requests.post(
    "https://<manageraddress>/api/admin/configuration/v1/worker_vm/",
    auth=('<user1>', '<password1>'),
    verify=False,
    stream=True,
    data=json.dumps({
        'name': 'new_node',
        'hostname': 'newnode',
        'domain': 'example.test',
        'address': '<newnode_ip_address>',
        'netmask': '<newnode_ip_mask>',
        'gateway': '<ip_gateway_address>',
        'password': '<newnode_password>',
        'node_type': 'PROXYING',
        'system_location': '/api/admin/configuration/v1/system_location/1/',
        'deployment_type': 'MANUAL',
        'vm_cpu_count': '8',
        'vm_system_memory': '16384',
    })
)
with open('conferencing_node.ova', 'wb') as handle:
    for chunk in response.iter_content(10*1024):
        handle.write(chunk)
print "Downloaded Conferencing Node OVA: conferencing_node.ova"
```

Deploying a Conferencing Node using a VM template and configuration file

You can use a `deployment_type` of `MANUAL-PROVISION-ONLY` to create a VM template for deployment onto unsupported hypervisors or orchestration layers.

```
import json
import requests
response = requests.post(
    "https://<manageraddress>/api/admin/configuration/v1/worker_vm/",
    auth=('<user1>', '<password1>'),
    verify=False,
    stream=True,
    data=json.dumps({
        'name': 'new_node',
        'hostname': 'newnode',
        'domain': 'example.test',
        'address': '<newnode_ip_address>',
        'netmask': '<newnode_ip_mask>',
        'gateway': '<ip_gateway_address>',
        'password': '<newnode_password>',
        'node_type': 'CONFERENCING',
        'system_location': '/api/admin/configuration/v1/system_location/1/',
        'deployment_type': 'MANUAL-PROVISION-ONLY',
    })
)
with open('conferencing_node.xml', 'wb') as handle:
    for chunk in response.iter_content(10*1024):
        handle.write(chunk)
print "Downloaded Conferencing Node provisioning document: conferencing_node.xml"
```

After the provisioning document has been obtained from the management API as above, it may be injected into the Conferencing Node to be provisioned as follows:

```
import requests
with open('conferencing_node.xml', 'rb') as handle:
    document = handle.read()
    response = requests.post(
        "https://<conferencingnodeaddress>:8443/configuration/bootstrap/",
        verify=False,
        headers={'Content-Type': 'text/xml'},
        data=document,
    )
    if response.status_code == requests.codes.ok:
        print "Successfully provisioned Conferencing Node"
```

Note that this API is available only on Conferencing Nodes created using the `MANUAL-PROVISION-ONLY` deployment type.

Uploading a service theme

The following example will create a new theme, upload the theme contents and then configure a VMR to use the theme:

```
import requests
import json
import urlparse
response = requests.post(
    "https://<manageraddress>/api/admin/configuration/v1/ivr_theme/",
    auth=('<user1>', '<password1>'),
    data=json.dumps({
        'name': 'New theme'
    }),
    verify=False)
theme_uri = response.headers['location']
theme_path = urlparse.urlsplit(theme_uri).path
response = requests.patch(
    theme_uri,
    auth=('<user1>', '<password1>'),
    files={
        'package': open('test_theme.zip', 'rb')
    },
    verify=False)
response = requests.patch(
    "https://<manageraddress>/api/admin/configuration/v1/conference/1/",
    auth=('<user1>', '<password1>'),
    data=json.dumps({
        'ivr_theme': theme_path
    }),
    verify=False)
```

Changing the distributed database setting of a Conferencing Node

The example below shows how to change the distributed database setting of a Conferencing Node. This enables the setting for the Conferencing Node with ID 1:

```
import requests
import json
response = requests.patch(
    "https://<manageraddress>/api/admin/configuration/v1/worker_vm/1/",
    auth=('<user1>', '<password1>'),
    verify=False,
    data=json.dumps({
        'enable_distributed_database': True
    })
)
print "Changed distributed database setting for Conferencing Node:", response.headers['location']
```

Returning a license

To return a license, send a DELETE request using the format:

```
import requests
response = requests.delete(
    "https://<manageraddress>/api/admin/configuration/v1/licence/<fulfillment_id>",
    auth=('<user1>', '<password1>'),
    verify=False
)
```

where:

- the fulfillment ID is a 10 digit number
- you can force offline mode by adding `?offline_mode=true` to the end of the URI, in which case it will return with a 202 response and a Location header telling you where to POST the response document.

Adding a license entitlement key

To add a license entitlement key, send a POST request using the format:

```
import json
import requests
response = requests.post(
    "https://<manageraddress>/api/admin/configuration/v1/licence/",
    auth=('<user1>', '<password1>'),
    verify=False,
    data=json.dumps({
        'entitlement_id' : '<entitlement key>'
    })
)
print "Added license entitlement key:", response.headers['location']
```

where:

- the entitlement key is in the form XXXX-XXXX-XXXX-XXXX (where X is a hex digit [0-9A-F])
- you can force offline mode by adding '**offline_mode**' : **true** to the JSON data, in which case it will return with a 202 response and a Location header telling you where to POST the response document.

Status API

The current status of the Pexip Infinity platform and any conference instances currently in progress can be viewed using the API.

Note that all date and time fields are in UTC time format.

Status resources

The following status resources are available via the REST API:

Component	Path
Conference instances	/api/admin/status/v1/conference/
Conference instances per node	/api/admin/status/v1/conference_shard/
Participants	/api/admin/status/v1/participant/
Registered aliases	/api/admin/status/v1/registration_alias/
Conferencing Nodes	/api/admin/status/v1/worker_vm/
Participant media statistics	/api/admin/status/v1/participant/<participant_id>/media_stream/
Conferencing Node load statistics	/api/admin/status/v1/worker_vm/<worker_vm_id>/statistics/
System locations	/api/admin/status/v1/system_location/
System location load statistics	/api/admin/status/v1/system_location/<system_location_id>/statistics/
Backplanes	/api/admin/status/v1/backplane/
Backplane media statistics	/api/admin/status/v1/backplane/<id>/media_stream/
Alarms	/api/admin/status/v1/alarm/
Licenses	/api/admin/status/v1/licensing/
Conference synchronization	/api/admin/status/v1/conference_sync/
List all cloud overflow Conferencing Nodes	/api/admin/status/v1/cloud_node/
List all locations monitored for dynamic bursting	/api/admin/status/v1/cloud_monitored_location/
List all locations that contain Conferencing Nodes that may be used for dynamic bursting	/api/admin/status/v1/cloud_overflow_location/
Exchange scheduler	/api/admin/status/v1/exchange_scheduler/

Specifying the object ID in the path

To retrieve the status of a specific resource, append the object ID of the resource to the path.

For example, a path of `/api/admin/status/v1/conference/68aef1a9-7b1b-4442-848d-cbad4b48b320/` retrieves the status of the conference with a conference ID of 68aef1a9-7b1b-4442-848d-cbad4b48b320.

You must also use the relevant object ID in the path of component requests when retrieving resources such as participant media statistics, Conferencing Node load statistics or backplane media statistics.

Resource details

More information can be obtained for each resource by downloading the resource schema in a browser. You may want to install a JSON viewer extension to your browser in order to view the JSON strings in a readable format.

For example, to view the schema for **conference instances** the URI would be:

```
https://<manageraddress>/api/admin/status/v1/conference/schema/?format=json
```

Each schema contains information on the available fields including:

- whether the field is optional (`nullable: true`) or required (`nullable: false`)
- the type of data the field must contain
- the choices for the field, e.g. `valid_choices: ["audio", "video", "video-only"]`
- which criteria can be used for [filtering](#) and [ordering](#) searches
- help text with additional information on usage.

Resource methods

Each status resource supports the following HTTP methods:

Method	Action
GET	Retrieves the current status for a resource.

Pagination and filtering

Status requests can be parameterized with pagination and filter fields. For more information, see [Retrieving, paginating, filtering and ordering resource details](#).

Examples

Getting all active conference instances

Retrieving all the conference instances is achieved by submitting a GET request to the resource URI for conference status:

```
import json
import requests
response = requests.get(
    "https://<manageraddress>/api/admin/status/v1/conference/",
    auth=('<user1>', '<password1>'),
    verify=False
)
print "Active conferences:", json.loads(response.text)['objects']
```

Example output

```
Active conferences: [
  {
    'start_time': '2015-04-02T09:46:06.106482',
    'resource_uri': '/api/admin/status/v1/conference/00000000-0000-0000-0000-000000000001/',
    'id': '00000000-0000-0000-0000-000000000001',
    'name': 'VMR_1',
    'service_type': 'conference',
    'is_locked': False,
    'tag': ''
  }
]
```

Getting all active Virtual Meeting Room conferences

Retrieving only those conference instances that are being held in a Virtual Meeting Room is achieved by submitting a GET request to the resource URI for conference status as above, but filtering it by `service_type`:

```
import json
import requests
response = requests.get(
    "https://<manageraddress>/api/admin/status/v1/conference/?service_type=conference",
    auth=('<user1>', '<password1>'),
    verify=False
)
print "Active conferences:", json.loads(response.text)['objects']
```

Getting all participants for a conference

Retrieving all the active participants for a conference instance is achieved by submitting a GET request to the resource URI for participant status and supplying a query parameter to specify the VMR.

The following example finds all participants for the Virtual Meeting Room **VMR_1**:

```
import json
import requests
response = requests.get(
    "https://<manageraddress>/api/admin/status/v1/participant/?conference=VMR_1",
    auth=('<user1>', '<password1>'),
    verify=False
)
print "Active participants for VMR_1:", json.loads(response.text)['objects']
```

Example output

```
Active participants for VMR_1: [
  {
    'bandwidth': 576,
    'bucketed_call_quality': '[0, 11, 1, 0, 0]',
    'call_direction': 'in',
    'call_quality': 'good',
    'call_uuid': '1c26be9c-6511-4e5c-9588-8351f8c3decd',
    'conference': 'VMR_1',
    'connect_time': '2015-04-02T09:46:11.116767',
    'conversation_id': '1c26be9c-6511-4e5c-9588-8351f8c3decd',
    'destination_alias': 'meet@example.com',
    'display_name': 'Alice',
    'encryption': 'On',
    'has_media': False,
    'historic_call_quality': '[1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1]',
    'id': '00000000-0000-0000-0000-000000000002',
    'is_muted': False,
    'is_on_hold': False,
    'is_presentation_supported': True,
    'is_presenting': False,
    'is_streaming': False,
    'license_count': 0,
    'license_type': 'nolicense',
    'media_node': '10.0.0.1',
    'parent_id': '',
    'participant_alias': 'Infinity_Connect_10.0.0.3',
    'protocol': 'WebRTC',
    'proxy_node': '10.10.0.46',
    'remote_address': '10.0.0.3',
    'remote_port': 54686,
    'resource_uri': '/api/admin/status/v1/participant/00000000-0000-0000-0000-000000000002/',
    'role': 'chair',
    'service_tag': '',
    'service_type': 'conference',
    'signalling_node': '10.0.0.1',
    'source_alias': 'Infinity_Connect_10.0.0.3',
    'system_location': 'London',
    'vendor': 'Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2272.101 Safari/537.36'
  },
  {
    'bandwidth': 768,
    'bucketed_call_quality': '[0, 7, 3, 1, 2]',
    'call_direction': 'in',
    'call_quality': 'good',
    'call_uuid': 'b0a5b554-d1de-11e3-a321-000c29e37602',
    'conference': 'VMR_1',
    'connect_time': '2015-04-02T09:46:53.712941',
    'conversation_id': 'b0a5b554-d1de-11e3-a321-000c29e37602',
    'destination_alias': 'meet@example.com',
    'display_name': 'Bob',
    'encryption': 'On',
    'has_media': False,
    'historic_call_quality': '[1, 1, 1, 1, 3, 4, 4, 2, 2, 2, 1, 1, 1]',
```

```
{
  'id': '00000000-0000-0000-0000-000000000003',
  'is_muted': False,
  'is_on_hold': False,
  'is_presentation_supported': False,
  'is_presenting': False,
  'is_streaming': False,
  'license_count': 1,
  'license_type': 'port',
  'media_node': '10.0.0.1',
  'parent_id': '',
  'participant_alias': 'bob@example.com',
  'protocol': 'H323',
  'proxy_node': '',
  'remote_address': '10.0.0.2',
  'remote_port': 11007,
  'resource_uri': '/api/admin/status/v1/participant/00000000-0000-0000-0000-000000000003/',
  'role': 'chair',
  'service_tag': '',
  'service_type': 'conference',
  'signalling_node': '10.0.0.1',
  'source_alias': 'bob@example.com',
  'system_location': 'London',
  'vendor': 'TANDBERG (Tandberg 257)'
}
```

Note that:

- Perceived call quality:
 - `historic_call_quality` shows the sequence of call quality calculations over time. The system looks at packet loss over multiple 20 second time windows throughout the call and calculates the call quality per window on the basis of < 1% packet loss is Good; < 3% is OK; < 10% is Bad; otherwise it is Terrible. These readings are reported as 0 = Unknown, 1 = Good, 2 = OK, 3 = Bad, 4 = Terrible.
 - `bucketed_call_quality` is a summary of the call quality calculations. For example, reading [0, 7, 3, 1, 2] from left to right means there were 0 x Unknown, 7 x Good, 3 x OK, 1 x Bad and 2 x Terrible quality calculations.
 - `call_quality` is the most frequently occurring call quality calculation in the last 3 windows.
- `conversation_id` is the same as `call_uuid` except for Lync / Skype for Business calls.
- `parent_id` is always "".

Getting the media statistics for a participant

Retrieving all the media stream statistics for a participant is achieved by submitting a GET request to the resource URI for the participant status.

The following example finds all media streams for the participant with ID 00000000-0000-0000-0000-000000000002.

```
import json
import requests
response = requests.get(
    "https://<manageraddress>/api/admin/status/v1/participant/00000000-0000-0000-0000-000000000002/media_stream/",
    auth=('<user1>', '<password1>'),
    verify=False
)
print "Media streams for participant:", json.loads(response.text)['objects']
```

Example output

```
Media streams for participant: [
  {
    'end_time': '',
    'id': '1',
    'node': '10.0.0.1'
    'rx_bitrate': 513,
    'rx_codec': 'VP8',
    'rx_jitter': 0.29,
    'rx_packet_loss': 0,
    'rx_packets_lost': 0,
    'rx_packets_received': 28761,
    'rx_resolution': '640x480',
    'start_time': '2015-07-22T13:13:52.921269',
    'tx_bitrate': 503,
    'tx_codec': 'VP8',
    'tx_jitter': 7.68,
    'tx_packet_loss': 0,
    'tx_packets_lost': 0,
    'tx_packets_sent': 26041,
    'tx_resolution': '768x448',
    'type': 'video'
  },
  {
    'end_time': '',
    'id': '0',
    'node': '10.0.0.1'
    'rx_bitrate': 12,
    'rx_codec': 'opus',
    'rx_jitter': 0.56,
    'rx_packet_loss': 0,
    'rx_packets_lost': 0,
    'rx_packets_received': 21148,
    'rx_resolution': '',
    'start_time': '2015-07-22T13:13:52.873617',
    'tx_bitrate': 2,
    'tx_codec': 'opus',
    'tx_jitter': 0.21,
    'tx_packet_loss': 0,
    'tx_packets_lost': 0,
    'tx_packets_sent': 42386,
    'tx_resolution': '',
    'type': 'audio'
  }
]
```

Getting the status of a Conferencing Node

By submitting a GET request to the status resource URI of a Conferencing Node you can get its current status. This status will show how an automatic deployment is progressing and the last time the Conferencing Node was configured and contacted.

```
import json
import requests
response = requests.get(
    "https://<manageraddress>/api/admin/status/v1/worker_vm/1/",
    auth=('<user1>', '<password1>'),
    verify=False
)
print "Conferencing node status:", json.loads(response.text)
```

Getting the load for a system location

By submitting a GET request to the statistics resource URI of the system location status you can get an estimate of the current media load.

The following example gets the media load for the system location with ID 1:

```
import json
import requests
response = requests.get(
    "https://<manageraddress>/api/admin/status/v1/system_location/1/statistics/",
    auth=('<user1>', '<password1>'),
    verify=False
)
print "System Location statistics:", json.loads(response.text)
```

Getting all registered aliases

Retrieving all the currently registered aliases is achieved by submitting a GET request to the resource URI for registration alias status:

```
import json
import requests
response = requests.get(
    "https://<manageraddress>/api/admin/status/v1/registration_alias/",
    auth=('<user1>', '<password1>'),
    verify=False
)
print "Registered aliases:", json.loads(response.text)['objects']
```

Listing all cloud overflow Conferencing Nodes

To retrieve a list of all cloud overflow Conferencing Nodes, submit a GET request to the resource URI for cloud node status:

```
import json
import requests
response = requests.get(
    "https://<manageraddress>/api/admin/status/v1/cloud_node/",
    auth=('<user1>', '<password1>'),
    verify=False
)
print "Cloud nodes:", json.loads(response.text)['objects']
```

Example output

```
Cloud nodes: {
  "meta": {
    "limit": 20,
    "next": null,
    "offset": 0,
    "previous": null,
    "total_count": 2
  },
  "objects": [
    {
      "aws_instance_id": "i-edblae65",
      "aws_instance_ip": "172.30.3.6",
```

```

    "aws_instance_launch_time": "2016-06-27T23:44:41",
    "aws_instance_name": "aws_overflow1",
    "aws_instance_state": "stopped",
    "cloud_instance_id": "i-edblae65",
    "cloud_instance_ip": "172.30.3.6",
    "cloud_instance_launch_time": "2016-06-27T23:44:41",
    "cloud_instance_name": "aws_overflow1",
    "cloud_instance_state": "stopped",
    "max_hd_calls": 0,
    "media_load": 100,
    "resource_uri": "/api/admin/status/v1/cloud_node/i-edblae65/",
    "workerm_configuration_id": 6,
    "workerm_configuration_location_name": "AWS",
    "workerm_configuration_name": "aws_overflow1"
  },
  {
    "aws_instance_id": "i-cfb0af47",
    "aws_instance_ip": "172.30.11.83",
    "aws_instance_launch_time": "2016-06-27T23:46:43",
    "aws_instance_name": "aws_overflow2",
    "aws_instance_state": "stopped",
    "cloud_instance_id": "i-cfb0af47",
    "cloud_instance_ip": "172.30.11.83",
    "cloud_instance_launch_time": "2016-06-27T23:46:43",
    "cloud_instance_name": "aws_overflow2",
    "cloud_instance_state": "stopped",
    "max_hd_calls": 0,
    "media_load": 100,
    "resource_uri": "/api/admin/status/v1/cloud_node/i-cfb0af47/",
    "workerm_configuration_id": 7,
    "workerm_configuration_location_name": "AWS",
    "workerm_configuration_name": "aws_overflow2"
  }
]
}

```

Note that:

- When a Conferencing Node is not available for use (in this example the instances are "stopped"), Pexip Infinity reports a media load of 100% to indicate that there is no current capacity available.
- The `aws_instance` prefixed fields are deprecated. Please use the `cloud_instance` prefixed fields instead.

Listing all locations monitored for dynamic bursting

To list all of the system locations that are being monitored for dynamic bursting, submit a GET request to the resource URI for cloud monitored location status:

```

import json
import requests
response = requests.get(
    "https://<manageraddress>/api/admin/status/v1/cloud_monitored_location/",
    auth=('<user1>', '<password1>'),
    verify=False
)
print "Monitored primary locations:", json.loads(response.text)['objects']

```

Example output

```

Monitored primary locations: {
  "meta": {
    "limit": 20,
    "next": null,
    "offset": 0,
    "previous": null,
    "total_count": 2
  },
  "objects": [
    {
      "free_hd_calls": 31,
      "id": 4,
      "max_hd_calls": 33,

```

```

    "media_load": 5,
    "name": "London",
    "resource_uri": "/api/admin/status/v1/cloud_monitored_location/4/"
  },
  {
    "free_hd_calls": 0,
    "id": 2,
    "max_hd_calls": 42,
    "media_load": 100,
    "name": "Oslo",
    "resource_uri": "/api/admin/status/v1/cloud_monitored_location/2/"
  }
]
}

```

Listing all locations containing dynamic bursting Conferencing Nodes

To list all of the system locations that contain Conferencing Nodes that may be used for dynamic bursting, submit a GET request to the resource URI for cloud monitored location status:

```

import json
import requests
response = requests.get(
    "https://<manageraddress>/api/admin/status/v1/cloud_overflow_location/",
    auth=('<user1>', '<password1>'),
    verify=False
)
print "Overflow locations:", json.loads(response.text)['objects']

```

Example output

```

Overflow locations: {
  "meta": {
    "limit": 20,
    "next": null,
    "offset": 0,
    "previous": null,
    "total_count": 2
  },
  "objects": [
    {
      "free_hd_calls": 31,
      "id": 4,
      "max_hd_calls": 33,
      "media_load": 5,
      "name": "London",
      "resource_uri": "/api/admin/status/v1/cloud_overflow_location/4/"
      "systemlocation_id": 3"
    },
    {
      "free_hd_calls": 0,
      "id": 2,
      "max_hd_calls": 42,
      "media_load": 100,
      "name": "Oslo",
      "resource_uri": "/api/admin/status/v1/cloud_overflow_location/2/"
      "systemlocation_id": 1"
    }
  ]
}

```


History API

The API can be used to view historical information about Pexip Infinity conference instances that are no longer in progress. For example, this can be used to obtain Call Detail Records (CDRs) of the Pexip Infinity platform.

Note that all date and time fields are in UTC time format.

Up to 10,000 conference instances are retained, along with all the participant instances associated with each of those conferences. Above 10,000 conference instances, each time a new entry is made the oldest entry is deleted (along with all the participant instances associated with it).

History resources

The following history resources are available via the REST API:

Component	Path
Alarm history *	/api/admin/history/v1/alarm/
Backplane	/api/admin/history/v1/backplane/
Backplane media statistics	/api/admin/history/v1/backplane/<backplane_id>/media_stream/
Conference instance	/api/admin/history/v1/conference/
Participant media statistics	/api/admin/history/v1/participant/<participant_id>/media_stream/
Participant	/api/admin/history/v1/participant/
Conferencing Node events *	/api/admin/history/v1/workervm_status_event/

* This resource is new in Pexip Infinity v16.

Resource details

More information can be obtained for each resource by downloading the resource schema in a browser. You may want to install a JSON viewer extension to your browser in order to view the JSON strings in a readable format.

For example, to view the schema for **backplane instances** the URI would be:

```
https://<manageraddress>/api/admin/history/v1/backplane/schema/?format=json
```

Each schema contains information on the available fields including:

- whether the field is optional (nullable: true) or required (nullable: false)
- the type of data the field must contain
- the choices for the field, e.g. valid_choices: ["audio", "video", "video-only"]
- which criteria can be used for [filtering](#) and [ordering](#) searches
- help text with additional information on usage.

Resource methods

Each history resource supports the following HTTP methods:

Method	Action
GET	Retrieves the history for a resource.

Pagination and filtering

History requests can be parameterized with pagination and filter fields. For more information, see [Retrieving, paginating, filtering and ordering resource details](#).

Examples

Getting all conference instances

Retrieving all the conference history is achieved by submitting a GET request to the resource URI for conference history:

```
import json
import requests
response = requests.get(
    "https://<manageraddress>/api/admin/history/v1/conference/",
    auth=('<user1>', '<password1>'),
    verify=False
)
print "Conference history:", json.loads(response.text)['objects']
```

Note that up to 10,000 conference instances may be stored in the history and by default the response is paginated.

Example output

```
Conference history: [
{
  'duration': 579,
  'end_time': '2015-04-10T09:49:26.556929',
  'id': '8583f400-7886-48c9-874b-5fefc2ac097e',
  'instant_message_count': 0,
  'name': 'meet.alice',
  'participant_count': 3,
  'participants': [
    '/api/admin/history/v1/participant/e9883f1d-88ca-495d-8366-b6eb772dfe57/',
    '/api/admin/history/v1/participant/5881adda-00ef-4315-8886-5d873d2ef269/',
    '/api/admin/history/v1/participant/29744376-0436-4fe1-ab80-06d93c71eb1c/'
  ],
  'resource_uri': '/api/admin/history/v1/conference/8583f400-7886-48c9-874b-5fefc2ac097e/',
  'service_type': 'conference',
  'start_time': '2015-04-10T09:39:47',
  'tag': ''
},
]
```

Getting all participants for a conference instance

Retrieving all the participants for a historical conference instance is achieved by submitting a GET request to the resource URI for participant history and supplying a query parameter to specify the conference.

The following example finds all participants for the conference with the ID 00000000-0000-0000-0000-000000000001:

```
import json
import requests
response = requests.get(
    "https://<manageraddress>/api/admin/history/v1/participant/?conference=00000000-0000-0000-0000-000000000001",
    auth=('<user1>', '<password1>'),
    verify=False
)
print "Participants for conference:", json.loads(response.text)['objects']
```

Example output

```
Participants for conference: [
{
  'bandwidth': 768,
  'bucketed_call_quality': '[0, 5, 2, 0, 0]',
  'call_direction': 'in',

```

```
'call_quality': 'good',
'call_uuid': 'b0a5b554-d1de-11e3-a321-000c29e37602',
'conference': '/api/admin/history/v1/conference/00000000-0000-0000-0000-000000000001/',
'conference_name': 'VMR_1',
'conversation_id': 'b0a5b554-d1de-11e3-a321-000c29e37602',
'disconnect_reason': 'Call disconnected',
'display_name': 'Bob',
'duration': 519,
'encryption': 'On',
'end_time': '2015-04-02T09:55:33.141261',
'has_media': True,
'historic_call_quality': '[1, 1, 2, 2, 1, 1, 1]',
'id': '00000000-0000-0000-0000-000000000003',
'is_streaming': false,
'license_count': 1,
'license_type': 'port',
'local_alias': 'meet@example.com',
'media_node': '10.0.0.1',
'media_streams': [
  {
    'end_time': '2015-07-22T12:43:33.645043',
    'id': 36,
    'node': '10.0.0.1',
    'participant': '/api/admin/history/v1/participant/5881adda-00ef-4315-8886-5d873d2ef269/',
    'rx_bitrate': 29,
    'rx_codec': 'opus',
    'rx_packet_loss': 0.0,
    'rx_packets_lost': 0,
    'rx_packets_received': 28091,
    'rx_resolution': '',
    'start_time': '2015-07-22T12:33:31.909536',
    'stream_id': '0',
    'stream_type': 'audio',
    'tx_bitrate': 2,
    'tx_codec': 'opus',
    'tx_packet_loss': 0.0,
    'tx_packets_lost': 0,
    'tx_packets_sent': 56347,
    'tx_resolution': ''
  },
  {
    'end_time': '2015-07-22T12:43:33.683385',
    'id': 37,
    'node': '10.0.0.1',
    'participant': '/api/admin/history/v1/participant/5881adda-00ef-4315-8886-5d873d2ef269/',
    'rx_bitrate': 511,
    'rx_codec': 'VP8',
    'rx_packet_loss': 0.01,
    'rx_packets_lost': 2,
    'rx_packets_received': 37027,
    'rx_resolution': '1280x720',
    'start_time': '2015-07-22T12:33:32.151438',
    'stream_id': '1',
    'stream_type': 'video',
    'tx_bitrate': 511,
    'tx_codec': 'VP8',
    'tx_packet_loss': 0.0,
    'tx_packets_lost': 0,
    'tx_packets_sent': 37335,
    'tx_resolution': '768x448'
  }
],
'parent_id': '29744376-0436-4fe1-ab80-06d93c71eb1c',
'protocol': 'WebRTC',
'proxy_node': 'None',
'remote_address': '10.0.0.2',
'remote_alias': 'Infinity_Connect_Media_10.0.0.2',
'remote_port': 11007,
'resource_uri': '/api/admin/history/v1/participant/00000000-0000-0000-0000-000000000003/',
'role': 'chair',
'service_tag': '',
'service_type': 'conference',
'signalling_node': '10.0.0.1',
```

```
'start_time': '2015-04-02T09:46:53.712941',
'system_location': 'London',
'vendor': 'Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2272.101
Safari/537.36'
},
{
'bandwidth': 0,
'bucketed_call_quality': '[]',
'call_direction': 'in',
'call_quality': 'unknown',
'call_uuid': '1c26be9c-6511-4e5c-9588-8351f8c3decd',
'conference': '/api/admin/history/v1/conference/00000000-0000-0000-0000-000000000001/',
'conference_name': 'VMR_1',
'conversation_id': '1c26be9c-6511-4e5c-9588-8351f8c3decd',
'disconnect_reason': 'Call disconnected',
'display_name': 'Alice',
'duration': 578,
'encryption': 'On',
'end_time': '2015-04-02T09:55:49.348317',
'has_media': False,
'historic_call_quality': '[]',
'id': '00000000-0000-0000-0000-000000000002',
'is_streaming': false,
'license_count': 0,
'license_type': 'nolicense',
'local_alias': 'meet@example.com',
'media_node': '10.0.0.1',
'media_streams': [],
'parent_id': '',
'protocol': 'WebRTC',
'proxy_node': 'None',
'remote_address': '10.0.0.2',
'remote_alias': 'Infinity_Connect_10.0.0.2',
'remote_port': 54686,
'resource_uri': '/api/admin/history/v1/participant/00000000-0000-0000-0000-000000000002/',
'role': 'chair',
'service_tag': '',
'service_type': 'conference',
'signalling_node': '10.0.0.1',
'start_time': '2015-04-02T09:46:11.116767',
'system_location': 'London',
'vendor': 'Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2272.101
Safari/537.36'
}
]
```

Note that:

- Perceived call quality:
 - `historic_call_quality` shows the sequence of call quality calculations over time. The system looks at packet loss over multiple 20 second time windows throughout the call and calculates the call quality per window on the basis of < 1% packet loss is Good; < 3% is OK; < 10% is Bad; otherwise it is Terrible. These readings are reported as 0 = Unknown, 1 = Good, 2 = OK, 3 = Bad, 4 = Terrible.
 - `bucketed_call_quality` is a summary of the call quality calculations. For example, reading [0, 7, 3, 1, 2] from left to right means there were 0 x Unknown, 7 x Good, 3 x OK, 1 x Bad and 2 x Terrible quality calculations.
 - `call_quality` is the most frequently occurring call quality calculation.
- `conversation_id` is the same as `call_uuid` except for Lync / Skype for Business calls.
- `parent_id` is always "".

Getting all participants for a time period

The following example uses filters to find all the participants whose call ended on 8 April 2015:

```
import json
import requests
response = requests.get(
    "https://<manageraddress>/api/admin/history/v1/participant/?end_time__gte=2015-04-08T00:00:00&end_
time__lt=2015-04-09T00:00:00",
    auth=('<user1>', '<password1>'),
    verify=False
)
print "Participants for 8th April :", json.loads(response.text)['objects']
```

Getting all participants with packet loss

The following example uses a filter to find all the participants with a media stream that had transmit packet loss greater than 0.5%:

```
import json
import requests
response = requests.get(
    "https://<manageraddress>/api/admin/history/v1/participant/?media_streams__tx_packet_loss__gte=0.5",
    auth=('<user1>', '<password1>'),
    verify=False
)
print "Participants with high transmit packet loss :", json.loads(response.text)['objects']
```

The following example uses a filter to find all the participants with a media stream that had receive packet loss greater than 0.5%:

```
import json
import requests
response = requests.get(
    "https://<manageraddress>/api/admin/history/v1/participant/?media_streams__rx_packet_loss__gte=0.5",
    auth=('<user1>', '<password1>'),
    verify=False
)
print "Participants with high receive packet loss :", json.loads(response.text)['objects']
```

Command API

The command API allows conference control and platform-level operations to be invoked.

Command resources

The following command resources are available via the REST API:

Component	Path
Dial	/api/admin/command/v1/participant/dial/
Disconnect participant	/api/admin/command/v1/participant/disconnect/
Disconnect conference	/api/admin/command/v1/conference/disconnect/
Mute participant	/api/admin/command/v1/participant/mute/
Mute all guests	/api/admin/command/v1/conference/mute_guests/
Unmute participant	/api/admin/command/v1/participant/unmute/
Unmute all guests	/api/admin/command/v1/conference/unmute_guests/
Lock conference	/api/admin/command/v1/conference/lock/
Unlock conference	/api/admin/command/v1/conference/unlock/
Unlock participant	/api/admin/command/v1/participant/unlock/
Transfer participant	/api/admin/command/v1/participant/transfer/
Create backup	/api/admin/command/v1/platform/backup_create/
Restore backup	/api/admin/command/v1/platform/backup_restore/
Sync LDAP template	/api/admin/command/v1/conference/sync/
Send provisioning email to VMR owner	/api/admin/command/v1/conference/send_conference_email/
Send provisioning email to device owner	/api/admin/command/v1/conference/send_device_email/
Certificate upload	/api/admin/command/v1/platform/certificates_import/
Start an overflow Conferencing Node	/api/admin/command/v1/platform/start_cloudnode/
Take system snapshot	/api/admin/command/v1/platform/snapshot/

Resource details

More information can be obtained for each resource by downloading the resource schema in a browser. You may want to install a JSON viewer extension to your browser in order to view the JSON strings in a readable format.

For example, to view the schema for the **dial** command the URI would be:

<https://<manageraddress>/api/admin/command/v1/participant/dial/schema/?format=json>

Each schema contains information on the available fields including:

- whether the field is optional (`nullable: true`) or required (`nullable: false`)
- the type of data the field must contain
- the choices for the field, e.g. `valid_choices: ["audio", "video", "video-only"]`
- help text with additional information on usage.

Resource methods

Each command resource supports the following HTTP methods:

Method	Action
POST	Invokes a new command for the resource.

Response format

The response for a command will be a JSON object with the following attributes:

Attribute	Value
status	<code>success</code> or <code>error</code> depending on whether or not the command succeeded.
message	An informational message string if the result was <code>error</code> .
data	Command-specific response data.

Individual commands may have response attributes specific to the command. See the command examples for more information.

Examples

Dialing a participant into a conference

By submitting a POST request to the `dial` resource URI, a new participant can be dialed into a conference.

When using this command, note that:

- The `conference_alias` is used for two purposes:
 - the participant being dialed will see the incoming call as coming from this alias
 - on answer, the participant will join the conference instance associated with this alias.
- One or both of the `node` and `system_location` must be specified.
- The `node` will override the `system_location` if both are given.
- If only the `system_location` is given, a random node in that location is used, which may just take the signaling for the call if the node is loaded.
- The `node` must be an IP address and not an FQDN.

The following example places a new call from a Conferencing Node in the **London** system location to the **H.323** endpoint with alias **alice**. Alice will see the call as coming from alias **meet@example.com**, and on answer will join the Virtual Meeting Room associated with that alias (in this case **VMR_1**, based on our configuration API examples), as a Host.

```
import requests
import json
response = requests.post(
    "https://<manageraddress>/api/admin/command/v1/participant/dial/",
    auth=('<user1>', '<password1>'),
    data={
        'conference_alias': 'meet@example.com',
        'destination': 'alice',
        'remote_display_name': 'Alice Parkes',
        'protocol': 'h323',
        'system_location': 'London',
        'role': 'chair',
    },
    verify=False)
print "New participant created:", json.loads(response.content)['data']['participant_id']
```

Note that if the dial command is successful the response will contain a `participant_id` attribute which can be used in queries for status and other conference control commands such as [Disconnecting a participant](#) and [Muting a participant](#).

Disconnecting a participant

By submitting a POST request to the `disconnect` resource URI, an existing participant can be disconnected from a conference instance.

```
import requests
response = requests.post(
    "https://<manageraddress>/api/admin/command/v1/participant/disconnect/",
    auth=('<user1>', '<password1>'),
    data={
        'participant_id': '00000000-0000-0000-0000-000000000001',
    },
    verify=False)
```


Muting a participant

By submitting a POST request to the `mute` resource URI, the audio being sent from an existing conference participant can be muted, meaning all other participants will not hear them.

```
import requests
response = requests.post(
    "https://<manageraddress>/api/admin/command/v1/participant/mute/",
    auth=('<user1>', '<password1>'),
    data={
        'participant_id': '00000000-0000-0000-0000-000000000001',
    },
    verify=False)
```

Muting all Guest participants

By submitting a POST request to the `mute_guests` resource URI, the audio being received from all Guest participants within an existing conference will be muted.

```
import requests
response = requests.post(
    "https://<manageraddress>/api/admin/command/v1/conference/mute_guests/",
    auth=('<user1>', '<password1>'),
    data={
        'conference_id': '00000000-0000-0000-0000-000000000001',
    },
    verify=False)
```

Unmuting a participant

By submitting a POST request to the `unmute` resource URI, a previously muted conference participant will have their audio restored, meaning other participants will again be able to hear them.

```
import requests
response = requests.post(
    "https://<manageraddress>/api/admin/command/v1/participant/unmute/",
    auth=('<user1>', '<password1>'),
    data={
        'participant_id': '00000000-0000-0000-0000-000000000001',
    },
    verify=False)
```

Unmuting all Guest participants

By submitting a POST request to the `unmute_guests` resource URI, the audio being received from all Guest participants within an existing conference will be unmuted.

```
import requests
response = requests.post(
    "https://<manageraddress>/api/admin/command/v1/conference/unmute_guests/",
    auth=('<user1>', '<password1>'),
    data={
        'conference_id': '00000000-0000-0000-0000-000000000001',
    },
    verify=False)
```

Locking a conference instance

By submitting a POST request to the conference `lock` resource URI, the conference instance will be locked, preventing new participants from joining. Instead new participants will be held at the **Waiting for conference host** screen. Note that if a service has a Host PIN, participants who enter the PIN will still be able to access the conference even when it is locked.

```
import requests
response = requests.post(
    "https://<manageraddress>/api/admin/command/v1/conference/lock/",
    auth=('<user1>', '<password1>'),
    data={
        'conference_id': '00000000-0000-0000-0000-000000000001',
    },
    verify=False)
```

Unlocking a conference instance

By submitting a POST request to the conference `unlock` resource URI, a previously locked conference instance can be unlocked, meaning new participants will be allowed to join. Any participants held at the **Waiting for conference host** screen will also automatically join the conference.

```
import requests
response = requests.post(
    "https://<manageraddress>/api/admin/command/v1/conference/unlock/",
    auth=('<user1>', '<password1>'),
    data={
        'conference_id': '00000000-0000-0000-0000-000000000001',
    },
    verify=False)
```

Unlocking a participant

By submitting a POST request to the participant `unlock` resource URI, a participant held at the **Waiting for conference host** screen because the conference has been locked will be allowed to join the conference.

```
import requests
response = requests.post(
    "https://<manageraddress>/api/admin/command/v1/participant/unlock/",
    auth=('<user1>', '<password1>'),
    data={
        'participant_id': '00000000-0000-0000-0000-000000000001',
    },
    verify=False)
```

Transferring a participant

By submitting a POST request to the participant `transfer` resource URI, a participant can be moved from one conference to another. The target conference is identified by an alias in the `conference_alias` field, and they will have the specified `role`.

If the target conference is PIN-protected, the participant will bypass the PIN entry.

```
import requests
response = requests.post(
    "https://<manageraddress>/api/admin/command/v1/participant/transfer/",
    auth=('<user1>', '<password1>'),
    data={
        'participant_id': '00000000-0000-0000-0000-000000000001',
        'conference_alias': 'meet@example.com',
        'role': 'guest'
    },
    verify=False)
```

Creating a system backup

You can create a system backup by submitting a POST request to the platform `backup_create` resource URI.

You must specify a passphrase (replacing `<backup_password>` in the example below). The passphrase is used to encrypt the backup file. You must remember the passphrase as it will be required if you need to subsequently restore the data.

```
import requests
response = requests.post(
    "https://<manageraddress>/api/admin/command/v1/platform/backup_create/",
    auth=('<user1>', '<password1>'),
    data={
        'passphrase': '<backup_password>',
    },
    verify=False)
```

The backup file is created on the Management Node under the `https://<manageraddress>/api/admin/configuration/v1/system_backup/` location.

Restoring a system backup

To restore a system backup (from a file stored on the Management Node) you need to:

1. List the available backups on the Management Node to identify the filename of the backup you want to restore.
2. Download the backup file to a temporary location.
3. Restore the contents of the backup file to your Pexip Infinity system.

Listing the available backups

You can perform a GET on `/api/admin/configuration/v1/system_backup/` to list the available backup files on the Management Node that can be restored (however, restoration must occur on exactly the same software version of Pexip Infinity that the backup was taken from).

```
import requests
response = requests.get(
    "https://<manageraddress>/api/admin/configuration/v1/system_backup/",
    auth=('<user1>', '<password1>'),
    verify=False)
print response.content
```

The response will include data similar to this for each backup file:

```
{
  "build": "26902.0.0",
  "date": "2016-01-28T15:36:16",
  "filename": "pexip_backup_10-44-7-0-mgr_11_26902.0.0_16_01_28_15_36_16.tar.gpg",
  "resource_uri": "/api/admin/configuration/v1/system_backup/pexip_backup_10-44-7-0-mgr_11_26902.0.0_16_01_28_15_36_16.tar.gpg/",
  "size": 9369536,
  "version": "11"
}
```

The "filename" value is what you will use in place of `<filename>` in the following GET command to download the file.

Note that if a backup file is no longer needed, you can perform a DELETE on the `https://<manageraddress>/api/admin/configuration/v1/system_backup/<filename>/` URL to delete individual files.

Downloading the backup file

Next you can perform a GET on `/api/admin/configuration/v1/system_backup/<filename>/` to download the backup file to a temporary location.

```
import requests
response = requests.get(
    "https://<manageraddress>/api/admin/configuration/v1/system_backup/<filename>/",
    auth=('<user1>', '<password1>'),
    verify=False)
with open("/tmp/temp.bak", "w") as file_handle:
    file_handle.write(response.content)
```

This downloads the backup file to `/tmp/temp.bak` — you can change this to your preferred location and filename if required.

Using the example response from the list of available backups, the GET would be for:

```
"https://10.44.7.0/api/admin/configuration/v1/system_backup/pexip_backup_10-44-7-0-mgr_11_26902.0.0_16_01_28_15_36_16.gpg/"
```

Restoring the backup

Finally, you can restore the contents of the backup file by submitting a POST request to the platform `backup_restore` resource URI, telling it to use the temporary file created in the previous step.

You must also specify the passphrase that was used to encrypt the backup file when it was generated.

```
import requests
response = requests.post(
    "https://<manageraddress>/api/admin/command/v1/platform/backup_restore/",
    auth=('<user1>', '<password1>'),
    data={
        'passphrase': '<backup_password>',
    },
    files={
        'package': open('/tmp/temp.bak'),
    },
    verify=False)
```

Starting an overflow Conferencing Node

If you are using dynamic bursting, the `start_cloudnode` resource can be used to manually start up a specific overflow node.

```
import requests
import json
response = requests.post(
    "https://<manageraddress>/api/admin/command/v1/platform/start_cloudnode/",
    data=json.dumps({'instance_id': '<node instance id>'}),
    auth=('<user1>', '<password1>'),
    verify=False)
print response.content
```

You can use the [cloud_node status resource](#) to obtain the instance IDs for your overflow nodes.

If the command is successful the response content takes the format: `'{"status": "success"}'`

If the Conferencing Node is already running, the response is: `'{"status": "failed", "error": "Unable to start a cloud node which isn't in 'STOPPED' state"}'`

If the `instance_id` does not match a cloud overflow node, the response is a 400 Bad Request with the following content: `'{"start_cloudnode": {"instance_id": ["Failed to find the cloud node."]}}'`

Retrieving, paginating, filtering and ordering resource details

This section describes how to retrieve the current configuration for a resource via the management API. It covers [Getting a single resource object](#), [Getting multiple resource objects](#), [Pagination](#), [Filtering](#) and [Ordering](#).

Getting a single resource object

By concatenating an object ID with the resource URI, details can be obtained for a single resource object.

For example, to GET the **alias** with ID **1** the URI would be:

```
https://<manageraddress>/api/admin/configuration/v1/conference_alias/1/
```

The response to a GET for a single resource will be a JSON object with attributes for each field of the resource.

Getting multiple resource objects

If a GET operation is invoked on the root resource URI then details about multiple objects are returned. The response to a GET for multiple resources is a JSON object with two attributes:

Attribute	Value
meta	This is an object giving meta information about the response.
objects	This is a list of resource objects.

For example, to GET all conference aliases the URI would be:

```
https://<manageraddress>/api/admin/configuration/v1/conference_alias/
```

Pagination

By default, the response is paginated and it contains the first page of 20 results. To retrieve subsequent pages of results the `offset` parameter must be specified in the URI.

To carry on from our previous example, to retrieve the second page of results the URI would be:

```
https://<manageraddress>/api/admin/configuration/v1/conference_alias/?offset=20
```

The `limit` parameter can be used to change the number of results in the response.

For example, to return the first 100 objects the URI would be:

```
https://<manageraddress>/api/admin/configuration/v1/conference_alias/?limit=100
```

Note that if there are a large number of aliases configured this may put a significant load on both the Management Node and the client making the request.

Filtering

A request for multiple objects can be filtered so that only those objects that match specific criteria are returned.

The following match specifiers are available:

Specifier	Description
exact	Matches objects whose field exactly matches the value.
ieexact	Case-insensitive version of the exact match.
contains	Matches objects whose field contains the value.
icontains	Case-insensitive version of the contains match.
startswith	Matches objects whose field starts with the value.
istartswith	Case-insensitive version of the startswith match.
endswith	Matches objects whose field ends with the value.
iendswith	Case-insensitive version of the endswith match.
regex	Matches objects whose field matches the regular expression value.
iregex	Case-insensitive version of the regex match.
lt	Matches objects whose field is less than the value.
lte	Matches objects whose field is less than or equal to the value.
gt	Matches objects whose field is greater than the value.
gte	Matches objects whose field is greater than or equal to the value.

The criteria that can be used for filtering are included at the end of the schema for each resource. If no **filtering** section is present, then filtering is not available.

For example, to see which criteria you can use to filter a search for service aliases, look at the associated schema:

```
https://<manageraddress>/api/admin/configuration/v1/conference_alias/schema/?format=json
```

You will see at the end:

```
filtering: {
  alias: 1,
  conference: 2,
  description: 1
}
```

This indicates that you can filter by any of the criteria listed above. The 1 and 2 following each criteria indicate that all of the match specifiers listed in the table above are valid when filtering using that criteria. Alternatively, if not all specifiers are valid, those that are valid are listed instead.

For example, to search for information about the alias **meet.alice** the URI would be:

```
https://<manageraddress>/api/admin/configuration/v1/conference_alias/?alias=meet.alice
```

For example, to search for all aliases that start with **meet.** the URI would be:

```
https://<manageraddress>/api/admin/configuration/v1/conference_alias/?alias__startswith=meet.
```

Ordering

A request for multiple objects can be ordered by field values.

For example, to get the conference history ordered by start time the URI would be:

```
https://<manageraddress>/api/admin/history/v1/conference/?order_by=start_time
```

The order can be reversed by adding a hyphen character (-) before the field name.

For example, to order by descending start time so that the most recent conferences are listed first the URI would be:

```
https://<manageraddress>/api/admin/history/v1/conference/?order_by=-start_time
```

The fields that can be used for ordering are included at the end of the schema for each resource. If no ordering section is present, then ordering is not available.

Using the API with SNMP

If SNMP is enabled on each Conferencing Node and the Management Node, you can use the Management API to obtain information using SNMP.

Note that SNMP is disabled by default, and is enabled and disabled on each node individually.

Examples

The examples below assume a community name of **public** (this is the default value and so is insecure).

Retrieving the SNMP sysName

```
# Retrieve the SNMPv2-MIB 'sysName' using an SNMP GET operation using SNMPV2c
from pysnmp.entity.rfc3413.oneliner import cmdgen
pexip_node_ip_address = '<pexipipaddress>'
cmdGen = cmdgen.CommandGenerator()
error_indication, error_status, error_index, var_binds = cmdGen.getCmd(
    cmdgen.CommunityData('public'),
    cmdgen.UdpTransportTarget((pexip_node_ip_address, 161)),
    cmdgen.MibVariable('SNMPv2-MIB', 'sysName', 0)
)
# Check for errors and print out results
if not error_indication and not error_status:
    for name, val in var_binds:
        print('%s = %s' % (name.prettyPrint(), val.prettyPrint()))
else:
    if error_indication:
        print(error_indication)
    if error_status:
        print('%s at %s' % (error_status.prettyPrint(), error_index and var_binds[int(error_index)-1]
or '?'))
```

Retrieving CPU load average

The value returned is on a scale where 1.00 equals one CPU core at full load. So a 4 CPU system would return values between 0.00 (completely idle) and 4.00 (completely maxed out).

```
# Retrieve the 1 minute CPU load average (OID: '.1.3.6.1.4.1.2021.10.1.3.1')
# using an SNMP GET operation using SNMPV2c
from pysnmp.entity.rfc3413.oneliner import cmdgen
pexip_node_ip_address = '<pexipipaddress>'
cmdGen = cmdgen.CommandGenerator()
error_indication, error_status, error_index, var_binds = cmdGen.getCmd(
    cmdgen.CommunityData('public'),
    cmdgen.UdpTransportTarget((pexip_node_ip_address, 161)),
    '.1.3.6.1.4.1.2021.10.1.3.1'
)
# Check for errors and print out results
if not error_indication and not error_status:
    for name, val in var_binds:
        print('%s = %s' % (name.prettyPrint(), val.prettyPrint()))
else:
    if error_indication:
        print(error_indication)
    if error_status:
        print('%s at %s' % (error_status.prettyPrint(), error_index and var_binds[int(error_index)-1]
or '?'))
```