Citrix NetScaler - NITRO Python - Getting Started Guide

Citrix® NetScaler® 10.5

Last Updated: March 2014

Document code: June 15 2014 23:52:15

Contents

NITRO API

The NetScaler NITRO protocol allows you to configure and monitor the NetScaler appliance programmatically.

NITRO exposes its functionality through Representational State Transfer (REST) interfaces. Therefore, NITRO applications can be developed in any programming language. Additionally, for applications that must be developed in Java or .NET or Python, NITRO APIs are exposed through relevant libraries that are packaged as separate Software Development Kits (SDKs).

Note: You must have a basic understanding of the NetScaler appliance before using NITRO.

To use the NITRO protocol, the client application needs only the following:

- Access to a NetScaler appliance, version 9.2 or later.
- To use REST interfaces, you must have a system to generate HTTP or HTTPS requests (payload in JSON format) to the NetScaler appliance. You can use any programming language or tool.
- For Java clients, you must have a system where Java Development Kit (JDK) 1.5 or later is available. The JDK can be downloaded from http://www.oracle.com/technetwork/java/javase/downloads/index.html.
- For .NET clients, you must have a system with .NET framework 3.5 or later installed. The .NET framework can be downloaded from http://www.microsoft.com/downloads/en/default.aspx.
- For Python clients, you must have a system with Python 2.7 or above version and the Requests library (available in <NITRO_SDK_HOME>/lib) installed.

Obtaining the NITRO Package

The NITRO package is available as a tar file on the **Downloads** page of the NetScaler appliance's configuration utility. You must download and un-tar the file to a folder on your local system. This folder is referred to as `<NITRO_SDK_HOME>` in this documentation.

The folder contains the NITRO libraries in the `lib` subfolder. The libraries must be added to the client application classpath to access NITRO functionality. The `<NITRO_SDK_HOME>` folder also provides samples and documentation that can help you understand the NITRO SDK.

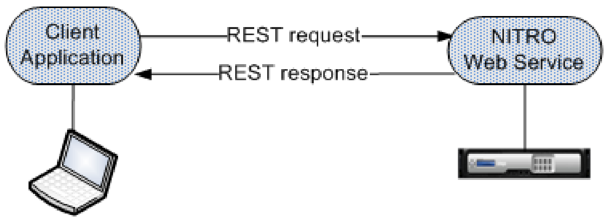
Note:

- The REST package contains only documentation for using the REST interfaces.
- For the Python SDK, the library must be installed on the client path. For installation instructions, read the `<NITRO_SDK_HOME>/README.txt` file.

How NITRO Works

The NITRO infrastructure consists of a client application and the NITRO Web service running on a NetScaler appliance. The communication between the client application and the NITRO web service is based on REST architecture using HTTP or HTTPS.

Figure 1-1. NITRO execution flow



As shown in the above figure, a NITRO request is executed as follows:

1. The client application sends REST request message to the NITRO web service. When using the SDKs, an API call is translated into the appropriate REST request message.
2. The web service processes the REST request message.
3. The NITRO web service returns the corresponding REST response message to the client application. When using the SDKs, the REST response message is translated into the appropriate response for the API call.

To minimize traffic on the NetScaler network, you retrieve the whole state of a resource from the server, make modifications to the state of the resource locally, and then upload it back to the server in one network transaction. For example, to update a load balancing virtual server, you must retrieve the object, update the properties, and then upload the changed object in a single transaction.

Note: Local operations on a resource (changing its properties) do not affect its state on the server until the state of the object is explicitly uploaded.

NITRO APIs are synchronous in nature. This means that the client application waits for a response from the NITRO web service before executing another NITRO API.

Python API

NetScaler NITRO APIs are categorized depending on the scope and purpose of the APIs into system APIs, feature configuration APIs, feature statistics APIs, and cluster APIs.

Additionally, you can import and export AppExpert applications. You can also troubleshoot NITRO operations.

Note: All NITRO operations are logged in the `/var/log/nitro.log` file on the appliance.

System APIs

The first step towards using NITRO is to establish a session with the NetScaler appliance and then authenticate the session by using the NetScaler administrator's credentials.

You must create an object of the `nssrc.com.citrix.netscaler.nitro.service.nitro_service` class by specifying the NetScaler IP (NSIP) address and the protocol to connect to the appliance (HTTP or HTTPS). You then use this object and log on to the appliance by specifying the user name and the password of the NetScaler administrator.

Note: You must have a user account on that appliance. The configuration operations that you perform are limited by the administrative roles assigned to your account.

The following sample code establishes a session with a NetScaler appliance with IP address 10.102.29.60 by using the HTTPS protocol:

```
//Specify the NetScaler appliance IP address and protocol
nitro_service ns_session = new
nitro_service("10.102.29.60","https")

//Specify the login credentials
ns_session.login("admin","verysecret")
```

Note: When using HTTPS, you must make sure that the root CA is added to the truststore. By default, NITRO validates the SSL certificate and verifies the hostname. To disable this validation, use the following:

```
ns_session.certvalidation = false
ns_session.hostnameverification = false
```

Note: By default, the connection to the appliance expires after 30 minutes of inactivity. You can modify the timeout period by specifying a new timeout period (in seconds) in the `login` method. For example, to modify the timeout period to 60 minutes:

```
ns_session.login("admin","verysecret",3600)
```

You must use the `nitro_service` object in all further NITRO operations on the appliance. For example to save the configurations on the appliance, you must use the `nitro_service` object as follows:

```
ns_session.save_config()
```

The `nitro_service` class also provides APIs to perform other system-level operations such as enabling and disabling NetScaler features and modes, saving and clearing NetScaler configurations, setting the session timeout, setting the severity of the exceptions to be handled, setting the behavior of bulk operations, and disconnecting from the appliance.

Feature Configuration APIs

NetScaler resources are organized into a set of packages or namespaces. Each package or namespace corresponds to a NetScaler feature. For example, all load-balancing related resources, such as load balancing virtual server, load balancing group, and load balancing monitor are available in `nssrc.com.citrix.netscaler.nitro.resource.config.lb`.

Similarly, all application firewall related resources, such as application firewall policy and application firewall archive are available in `nssrc.com.citrix.netscaler.nitro.resource.config.appfw`.

Each NetScaler resource is represented by a class. For example, the class that represents a load balancing virtual server is called `lbvserver` (in `nssrc.com.citrix.netscaler.nitro.resource.config.lb`). The state of a resource is represented by properties of a class. You can get and set the properties of the class.

Note: The setter and getter properties are always executed locally on the client. They do not involve any network interaction with the NITRO web service. All properties have basic simple types: integer, long, boolean, and string.

A resource class provides APIs to perform the following operations:

Create | Retrieve | Update | Delete | Enable/Disable | Unset | Bind/Unbind | Global bind | Bulk operations

Create

To create a new resource, instantiate the resource class, configure the resource by setting its properties locally, and then upload the new resource instance to the NetScaler appliance.

The following sample code creates a load balancing virtual server:

```
//Create an instance of the lbvserver class
lbvserver new_lbvserver_obj = new lbvserver()

//Set the properties of the resource locally
```

```
new_lbvserver_obj.name = "MyFirstLbVServer"
new_lbvserver_obj.ipv46 = "10.102.29.88"
new_lbvserver_obj.port = 88
new_lbvserver_obj.servicetype = "HTTP"
new_lbvserver_obj.lbmethod = "ROUNDROBIN"

//Upload the resource to NetScaler
lbvserver.add(ns_session, new_lbvserver_obj)
```

Retrieve

To retrieve the properties of a resource, you retrieve the resource object from the NetScaler appliance. Once the object is retrieved, you can extract the required properties of the resource locally, without further network traffic.

The following sample code retrieves the details of a load balancing virtual server:

```
//Retrieve the resource object from the NetScaler
new_lbvserver_obj =
lbvserver.get(ns_session,"MyFirstLbVServer")

//Extract the properties of the resource from the object
locally
print(new_lbvserver_obj.name)
print(new_lbvserver_obj.servicetype)
```

You can also retrieve resources by specifying a filter on the value of their properties by using the `nssrc.com.citrix.netscaler.nitro.util.filtervalue` class.

For example, you can retrieve all the load balancing virtual servers that have their port set to 80 and servicetype to HTTP:

```
filtervalue[] filter = new filtervalue[2]
filter[0] = new filtervalue("port","80")
filter[1] = new filtervalue("servicetype","HTTP")
lbvserver[] result = lbvserver.filtered(ns_session, filter)
```

You can also retrieve all NetScaler resources of a certain type, such as all services in the NetScaler appliance, by calling the static `get()` method on the service class, without providing a second parameter, as follows:

```
service[] resources = service.get(ns_session)
```

Update

To update the properties of a resource, instantiate the resource class, specify the name of the resource to be updated, configure the resource by updating its properties locally, and then upload the updated resource instance to the NetScaler appliance.

The following sample code updates the service type and load balancing method of a load balancing virtual server:

```
//Create an instance of the lbvserver class
lbvserver update_lb = new lbvserver()
```

```
//Specify the name of the lbvserver to be updated
update_lb.name = "MyFirstLbVServer"

//Specify the updated service type and lb method
update_lb.servicetype = "https"
update_lb.lbmethod = "LEASTRESPONSETIME"

//Upload the resource to NetScaler
lbvserver.update(ns_session, update_lb)
```

Note: Some properties in some NetScaler resources are not allowed to be modified after creation. The port number or the service type (protocol) of a load balancing virtual server or a service, are examples of such properties. Even though the update method appears to succeed, these properties retain their original values on the appliance.

Delete

To delete an existing resource, invoke the `delete()` method on the resource class, by passing the name of the resource.

The following sample code deletes a load balancing virtual server with name "MyFirstLbVServer":

```
lbvserver remove_lb = new lbvserver()
remove_lb.name = "MyFirstLbVServer"
lbvserver.delete(ns_session, remove_lb)
```

Enable/Disable

To enable a resource, invoke the `enable()` method.

The following sample code enables a load balancing virtual server named "lb_vip":

```
lbvserver obj = new lbvserver()
obj.name = "lb_vip"
lbvserver.enable(ns_session, obj)
```

Note: To disable a resource, invoke the `disable()` method.

```
lbvserver.disable(ns_session, obj)
```

Unset

To unset the value that is set to a parameter, invoke the `unset()` method on the resource class, by passing the name of the resource and the parameters to be unset. If the parameter has a default value, the value is reset to that value.

The following sample code unsets the load balancing method and the comments of a load balancing virtual server named "lb_123":

```
lbvserver lb1 = new lbvserver()
lb1.name = "lb_123"
String args[] = {"comment", "lbmethod"}
lbvserver.unset(ns_session, lb1, args)
```

Bind/Unbind

NetScaler resources form relationships with each other through the process of binding. This is how services are associated with a load balancing virtual server (by binding them to it), or how various policies are bound to a load balancing virtual server. Each binding relationship is represented in NITRO by its own class.

To bind one NetScaler resource to another, you must instantiate the appropriate binding class (for example, to bind a service to a load balancing virtual server, you must instantiate the `lbvserver_service_binding` class) and add it to the NetScaler configuration (by using the static `add()` method on this class).

Binding classes have a property representing the name of each resource in the binding relationship. They can also have other properties related to that relationship (for example, the weight of the binding between a load balancing virtual server and a service).

The following sample code binds a service to a load balancing virtual server, by specifying a certain weight for the binding:

```
lbvserver_service_binding bindObj = new
lbvserver_service_binding()
bindObj.name = "MyFirstLbVServer"
bindObj.servicename = "svc_prod"
bindObj.weight = 20
lbvserver_service_binding.add(ns_session, bindObj)
```

Note: To unbind a resource from another, invoke the `delete()` method from the resource binding class, by passing the name of the two resources.

The following code sample unbinds a service from a server:

```
lbvserver_service_binding bindObj = new
lbvserver_service_binding()
bindObj.name = "MyFirstLbVServer"
bindObj.servicename = "svc_prod"
lbvserver_service_binding.delete(ns_session, bindObj)
```

Global bind

Some NetScaler resources can be bound globally to affect the whole system. For example, a compression policy can be bound to an load balancing virtual server, in which case the policy affects only the traffic on that load balancing virtual server.

However, if bound globally, it can affect any traffic on the appliance, regardless of which virtual servers handle the traffic.

Some NITRO classes can be used to bind resources globally. These classes have names that follow the following pattern:
`<featurename>global_<resourcetype>_binding`.

For example, the class `aaaglobal_preauthenticationpolicy_binding` is used to bind preauthentication policies globally.

The following sample code creates a preauthentication action and a preauthentication policy that uses that action, and then binds the policy globally at priority 200:

```
aaapreauthenticationaction preauth_act1
aaapreauthenticationpolicy preauth_pol1
aaaglobal_aaapreauthenticationpolicy_binding glob_binding
preauth_act1 = new aaapreauthenticationaction()
preauth_act1.name = "preauth_act1"
preauth_act1.preauthenticationaction = "ALLOW"
aaapreauthenticationaction.add(ns_session, preauth_act1)

preauth_pol1 = new aaapreauthenticationpolicy()
preauth_pol1.name = "preauth_pol1"
preauth_pol1.rule = "CLIENT.APPLICATION.PROCESS(antivirus.exe)
EXISTS"
preauth_pol1.reqaction = "preauth_act1"
aaapreauthenticationpolicy.add(ns_session, preauth_pol1)

glob_binding = new
aaaglobal_aaapreauthenticationpolicy_binding()
glob_binding.policy = "preauth_pol1"
glob_binding.priority = 200
aaaglobal_aaapreauthenticationpolicy_binding.add(ns_session,
glob_binding)
```

Bulk operations

You can create, retrieve, update, and delete multiple resources simultaneously and thus minimize network traffic. For example, you can add multiple load balancing virtual servers in the same operation. To perform a bulk operation, you instantiate an array of the resource class, configure the properties of all the instances locally, and then upload all the instances to the NetScaler with one command.

To account for the failure of some operations within the bulk operation, NITRO allows you to configure one of the following behaviors:

- **Exit.** When the first error is encountered, the execution stops. The commands that were executed before the error are committed.
- **Rollback.** When the first error is encountered, the execution stops. The commands that were executed before the error are rolled back. Rollback is only supported for add and bind commands.
- **Continue.** All the commands in the list are executed even if some commands fail.

Note: You must configure the required behavior while establishing a connection with the appliance.

```
nitro_service ns_session = new
nitro_service("10.102.29.60","http")
ns_session.onerror = OnerrorEnum.CONTINUE
ns_session.login("admin","verysecret")
```

The following sample code creates two load balancing virtual servers:

```
//Create an array of lbvserver instances
lbvserver[] lbs = new lbvserver[2]

//Specify properties of the first lbvserver
lbs[0] = new lbvserver()
lbs[0].name = "lbvserv1"
lbs[0].servicetype = "http"
lbs[0].ipv46 = "10.70.136.5"
lbs[0].port = 80

//Specify properties of the second lbvserver
lbs[1] = new lbvserver()
lbs[1].name = "lbvserv2"
lbs[1].servicetype = "https"
lbs[1].ipv46 = "10.70.136.5"
lbs[1].port = 443

//Upload the properties of the two lbvservers to the NetScaler
lbvserver.add(ns_session, lbs)
```

Cluster APIs

For managing clusters, you can add or remove a cluster instance or an individual node and perform a few other instance or node operations such as viewing instance or node properties. You can also configure the cluster IP address. Other cluster-management tasks include joining a NetScaler appliance to the cluster and configuring a linkset.

Cluster Instance Operations

The `nssrc.com.citrix.netscaler.nitro.resource.config.cluster.clusterinstance` class provides APIs to manage a cluster instance.

The following sample code creates a cluster instance with ID 1:

```
clusterinstance new_cl_inst_obj = new clusterinstance()

//Set the properties of the cluster instance locally
new_cl_inst_obj.clid = 1
new_cl_inst_obj.preemption = "ENABLED"
```

```
//Upload the cluster instance
clusterinstance.add(ns_session, new_cl_inst_obj)
```

Cluster Node Operations

The `nssrc.com.citrix.netscaler.nitro.resource.config.cluster.clusternode` class provides APIs to manage cluster nodes.

The following sample code adds a cluster node with NSIP address 10.102.29.60:

```
clusternode new_cl_node_obj = new clusternode()

//Set the properties of the cluster node locally
new_cl_node_obj.nodeid = 0
new_cl_node_obj.ipaddress = "10.102.29.60"
new_cl_node_obj.state = "ACTIVE"
new_cl_node_obj.backplane = "0/1/1"

//Upload the cluster node
clusternode.add(ns_session, new_cl_node_obj)
```

Add a Cluster IP Address

The `nssrc.com.citrix.netscaler.nitro.resource.config.ns.nsip` class provides the `add()` API to configure an IP address. To configure the IP address as a cluster IP address, you must specify the type as CLIP.

The following sample code configures a cluster IP address on NetScaler appliance with IP address 10.102.29.60:

```
nsip new_nsip_obj = new nsip()

//Set the properties locally
new_nsip_obj.ipaddress = "10.102.29.61"
new_nsip_obj.netmask = "255.255.255.255"
new_nsip_obj.type = "CLIP"

//Upload the cluster node
nsip.add(ns_session, new_nsip_obj)
```

Add a Spotted IP Address

The `nssrc.com.citrix.netscaler.nitro.resource.config.ns.nsip` class provides the `add()` API to configure an IP address. To configure the IP address as spotted, you must specify the ID of the node that must own the IP address. This configuration must be done on the cluster IP address.

The following sample code configures a spotted SNIP address on a node with ID 1:

```
nsip new_nsip_obj = new nsip()

//Set the properties locally
new_nsip_obj.ipaddress = "10.102.29.77"
new_nsip_obj.netmask = "255.255.255.0"
new_nsip_obj.type = "SNIP"
new_nsip_obj.ownernode = 1

//Upload the cluster node
nsip.add(ns_session, new_nsip_obj)
```

Join NetScaler Appliance to Cluster

The `nssrc.com.citrix.netscaler.nitro.resource.config.cluster.cluster` class provides the `join()` API to join a NetScaler appliance to the cluster. You must specify the cluster IP address and the nsroot password of the configuration coordinator.

The following sample joins a NetScaler appliance to a cluster:

```
cluster new_cl_obj = new cluster()

//Set the properties of the cluster  locally
new_cl_obj.clip = "10.102.29.61"
new_cl_obj.password = "verysecret"

//Upload the cluster
cluster.add(ns_session, new_cl_obj)
```

Linkset Operations

The `nssrc.com.citrix.netscaler.nitro.resource.config.network.linkset` class provides the APIs to manage linksets.

To configure a linkset, do the following:

1. Add a linkset by invoking the `add()` method of the `linkset` class.
2. Bind the interfaces to the linkset using the `add()` method of the `linkset_interface_binding` class.

The following sample code creates a linkset LS/1 and bind interfaces 1/1/2 and 2/1/2 to it:

```
//Create the linkset
linkset new_linkset_obj = new linkset()
new_linkset_obj.id = "LS/1"
linkset.add(ns_session, new_linkset_obj)

//Bind the interfaces to the linkset
linkset_interface_binding new_linkif_obj = new
linkset_interface_binding()
new_linkif_obj.id = "LS/1"
new_linkif_obj.ifnum = "1/1/2 2/1/2"
linkset_interface_binding.add(ns_session, new_linkif_obj)
```

Feature Statistics APIs

The NetScaler appliance collects statistics about the usage of its features and the corresponding resources. You can retrieve these statistics by using NITRO API. The statistics APIs are available in different packages from the configuration APIs.

The APIs to retrieve statistics of NetScaler features are available in packages that have the following pattern: `nssrc.com.citrix.netscaler.nitro.resource.stat.<feature>`.

For example, APIs to retrieve statistics of the load balancing virtual server are available in the `nssrc.com.citrix.netscaler.nitro.resource.stat.lb` package.

The following sample code retrieves the statistics of a load balancing virtual server and displays some of the statistics returned:

```
lbvserver_stats stats =
lbvserver_stats.get(ns_session,"MyFirstLbVServer")
print(stats.curclntconnections)
print(stats.deferredreqrate)
```

Note: Not all NetScaler features and resources have statistic objects associated with them.

AppExpert Application APIs

To export an AppExpert application, you must instantiate the `nssrc.com.citrix.netscaler.nitro.resource.config.app.application` class, configure the properties of the AppExpert locally, and then export the AppExpert application.

The following sample code exports an AppExpert application named "MyApp1":

```
application myapp = new application()
myapp.appname = "MyApp1"
myapp.apptemplatefilename = "myapp_template"
application.export(ns_session, myapp)
```

You can also import an AppExpert application. You must instantiate the `nssrc.com.citrix.netscaler.nitro.resource.config.app.application` class, configure the properties of the AppExpert locally, and then import the AppExpert application.

The following sample code imports an AppExpert application named "MyApp1":

```
application myapp = new application()
myapp.appname = "MyApp1"
myapp.apptemplatefilename = "myapp_template"
application.Import(ns_session, myapp)
```

Exception Handling

The status of a NITRO request is captured in the `nssrc.com.citrix.netscaler.nitro.exception.nitro_exception` class. This class provides the following details of the exception:

- **Session ID.** The session in which the exception occurred.
- **Severity.** The severity of the exception: error or warning. By default, only errors are captured. To capture warnings, you must set the warning flag to true, while connecting to the appliance.
- **Error code.** The status of the NITRO request. An error code of 0 indicates that the NITRO request is successful. A non-zero error code indicates an error in processing the NITRO request.
- **Error message.** Provides a brief description of the exception.

For a list of error codes, see the `errorlisting.html` file available in the `<NITRO_SDK_HOME>/doc/api_reference` folder.

Unsupported NetScaler Operations

Some NetScaler operations that are available through the command line interface and through the configuration utility, are not available through NITRO APIs. The following list provides the NetScaler operations not supported by NITRO:

- install API
- diff API on nsconfig resource
- UI-internal APIs (update, unset, and get)
- show ns info
- Application firewall APIs:
  - `importwsdl`
  - `importcustom`
  - `importxmlschema`
  - `importxmlerrorpage`
  - `importhtmlerrorpage`
  - `rmwsdl`
  - `rmcustom`
  - `rmxmlschema`
  - `rmxmlerrorpage`
  - `rmhtmlerrorpage`

- CLI-specific APIs:
  - `ping`
  - `ping6`
  - `traceroute`
  - `traceroute6`
  - `nstrace`
  - `scp`
  - `configaudit`
  - `show defaults`
  - `show permission`
  - `batch`
  - `source`