Automate NetScaler with Ansible

Introduction: Networking Automation for Digital Transformation

Leading-edge businesses are changing the way they respond to change. Their goal is to react quickly to customer input and market data by rapidly enhancing software applications and optimizing networks.

But change is risky, especially in complex environments that send multiple gigabits of data every second to mission-critical applications. Traditional manual methods for configuring networking devices use command line interfaces and GUIs. These are prone to human errors that can lead to performance problems and downtime.

The solution is automation and orchestration. Automation makes individual changes to software and networks repeatable and reliable. Orchestration coordinates multiple changes so major projects, such as rolling out new applications and upgrading infrastructure, can be accomplished quickly and safely.

This white paper looks at how the Ansible can help organizations automate the management of Citrix NetScaler appliances. We will examine some of the characteristics of the Ansible tool, highlight a few of the benefits, and review three Ansible playbooks to see how they automate three processes:

- Deploying a NetScaler Application Delivery Controller (ADC) as part of a new web application rollout
- Updating an application service without disrupting users
- Configuring global load balancing to improve application performance and availability

Overview of Ansible

Ansible is an open-source tool for automating the deployment and upgrading of applications, and the configuration of software for networking and security. It uses a simple, declarative automation language to describe the steps in each process. The steps are combined in "playbooks" that execute on the Ansible automation engine.

Playbooks can be made up of "modules" that automate specific tasks. Ansible modules can be used as building blocks and recombined to orchestrate changes to complex, multi-tier applications. Over 1,300 pre-tested modules are available online, including several for managing NetScaler capabilities (see links below).

Ansible communicates with NetScaler devices using a REST API (called NITRO). It does not require any agents or additional software on devices it manages.

More information about Ansible

The Ansible web site: https://www.ansible.com/
The Ansible documentation web site: http://docs.ansible.com/
Ansible modules for NetScaler: http://docs.ansible.com/ansible/latest/list_of_network_modules.html#netscaler

The Benefits of Using Ansible with NetScaler

For application development teams and DevOps

Application development teams and DevOps need to stage, deploy, and update complex, multi-tier applications. NetScaler application delivery controllers are a critical part of many of those applications. They provide critical services such as load balancing, application acceleration, context-aware application switching, DoS protection, application firewalls, and features for high availability. By automating the provisioning and configuration of NetScaler appliances, Ansible helps development teams and DevOps roll out and enhance applications faster, without disrupting ongoing business processes.

For network and IT operations groups

Network administrators and IT operations groups need to continuously tune and improve network performance, while providing very near to 100 percent availability. Networking devices like NetScaler are among their key tools for ensuring fast, predictable networks. Administrators can use Ansible to automate hundreds of configuration changes to NetScaler appliances. This eliminates manual configuration processes, facilitates rapid improvements to network performance, and dramatically reduces human errors.

So everyone can work together

Ansible for NetScaler helps application development teams and IT operations groups work together. Because Ansible's automation language and playbooks are human readable and easy to learn, both organizations can participate in automating NetScaler updates. Also, Ansible's building block approach to creating playbooks greatly simplifies the task of orchestrating NetScaler configuration tasks as part of large, complex application deployment and infrastructure upgrade projects.

Example #1: Deploying a NetScaler ADC as Part of a New Web Application Rollout

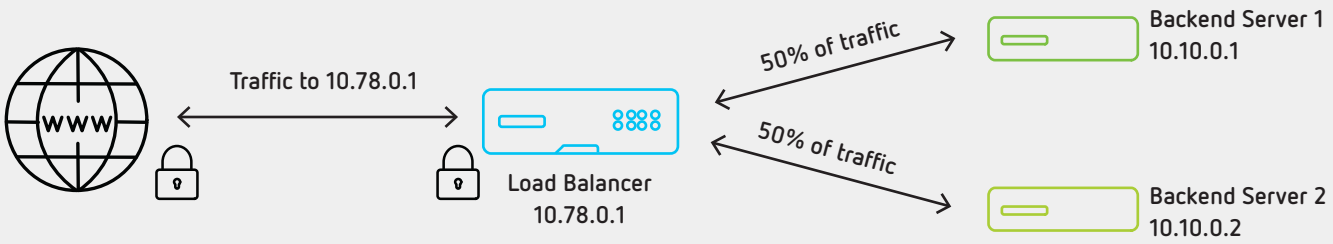
A new customer-facing web application is being deployed. It will run on two web servers in the same data center.

A NetScaler appliance will be placed between the web and the two servers. The NetScaler node will:

- Load balance traffic equally between the two backend servers.
- Ensure HTTPS encryption between the clients on the network and the NetScaler appliance.

NetScaler Load Balancer

A NetScaler appliance load balancing traffic across servers.



Let's see how Ansible can be used for this basic NetScaler HTTPS setup. While it would not be difficult to perform this effort manually once, creating the Ansible playbook allows the work to be replicated tens or hundreds of times, with very little effort and consistent results.

Preparation

Ansible 2.4 and the NITRO Python SDK must be installed on the machine used to run the Ansible playbooks. This machine must be able to communicate with the REST API ports (80/443) on the target NetScaler devices.

To install the NITRO Python SDK, clone the NetScaler Ansible modules in the GitHub repository (https://github.com/citrix/netscaler-ansible-modules).

From the checkout directory, run: pip install deps/nitro-python-1.0_oban_51_11.tar.gz

Alternately, you can find the NITRO Python SDK from the "Downloads" tab on the NetScaler GUI.

Also, you will also need to define a host in your Ansible inventory file. The file must have a NetScaler host configured, along with the needed connection variables.

Here is a sample inventory file:
netscaler nsip=10.78.60.200 nitro_user=nsroot nitro_pass=nsroot

nsip is the IP address of the NetScaler node that will receive the NITRO API calls. Nitro_user and nitro_pass are the credentials used to authenticate to the NetScaler appliance. Here, their initial default values are "nsroot."

The playbook

The example playbook for this scenario performs three tasks. It:

1. Sets up a service group with two backend services. These services are given equal weights, so the NetScaler load balancer will direct 50 percent of the requests to each service.
2. Sets up an SSL certificate key that will be used to encrypt traffic between the network web clients and the NetScaler node.
3. Sets up the load balance virtual server that will bind together the previous elements.

Notes:

- The files containing the SSL keys must already be present on the NetScaler node.
- The "gather_facts" playbook parameter should be set to "no."
- The "delegate_to: localhost" parameter should be set for each task.

Ansible will not connect directly to the NetScaler node via SSH, because the Ansible NetScaler modules configure NetScaler appliance using the NITRO REST API.

Here is the playbook.

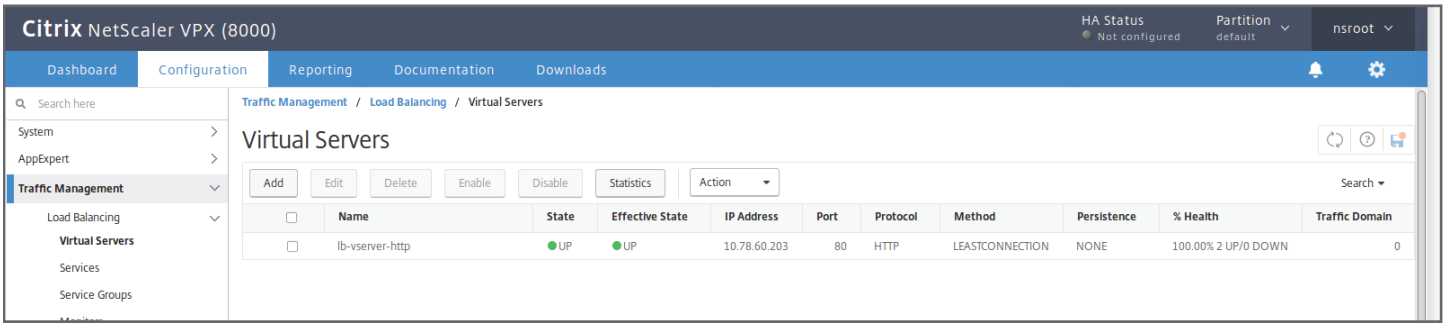
```
- hosts: netscaler
  gather_facts: no
  tasks:
   - name: Setup http service group
     delegate_to: localhost
     netscaler_servicegroup:
       nsip: "{{ nsip }}"
       nitro_user: "{{ nitro_user }}"
       nitro_pass: "{{ nitro_pass }}"

       servicegroupname: service-group-http
       servicetype: HTTP
       maxclient: 4000
       servicemembers:
          - ip: 10.10.0.1
            port: 80
            weight: 50
          - ip: 10.10.0.2
            port: 80
            weight: 50

   - name: Setup ssl certificate key
     delegate_to: localhost
     netscaler_ssl_certkey:
       nsip: "{{ nsip }}"
       nitro_user: "{{ nitro_user }}"
       nitro_pass: "{{ nitro_pass }}"
       certkey: certificate_http
       cert: server.crt
       key: server.key

   - name: Setup lb vserver
     delegate_to: localhost
     netscaler_lb_vserver:
       nsip: "{{ nsip }}"
       nitro_user: "{{ nitro_user }}"
       nitro_pass: "{{ nitro_pass }}"

       name: lb-vserver-http
       servicetype: SSL
       ipv46: 10.78.0.1
       port: 443
       ssl_certkey: certificate_http
       servicegroupbindings:
          - servicegroupname: service-group-http
```

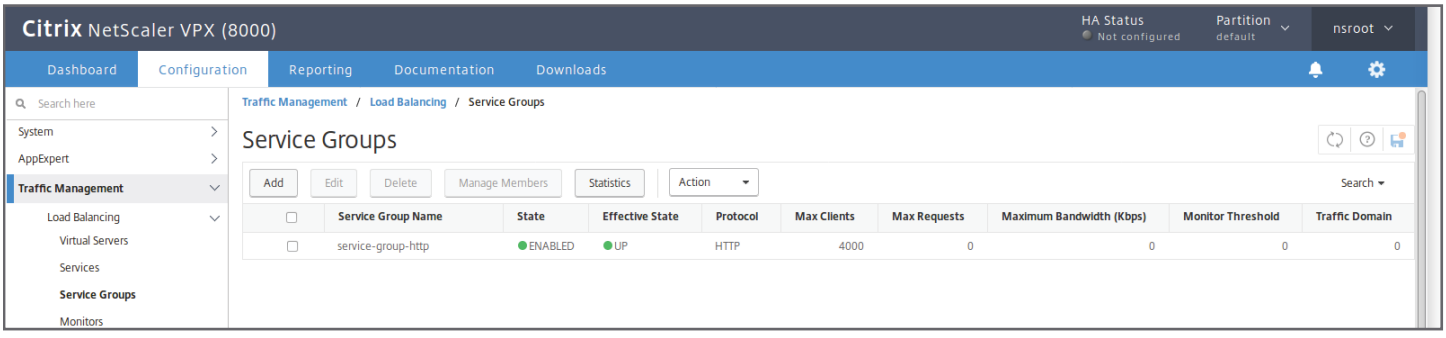
Output

Here is the output of the playbook run. It shows that all tasks completed successfully (ok=3), and that all three of the tasks run made changes to the NetScaler configuration (changed=3).

```
PLAY [netscaler] ***************************************************************************************************************************
*****************************

TASK [Setup http service group] ************************************************************************************************************
*****************************
changed: [netscaler -> localhost]

TASK [Setup ssl certificate key] ***********************************************************************************************************
*************************
changed: [netscaler -> localhost]

TASK [Setup lb vserver] ********************************************************************************************************************
*****************************
changed: [netscaler -> localhost]

PLAY RECAP *********************************************************************************************************************************
*******************************
netscaler                  : ok=3    changed=3    unreachable=0    failed=0
```

The NetScaler screens below indicate that the servicegroup and the lb vserver have been set up according to our specification





The new service is now accessible at the configured IP address.

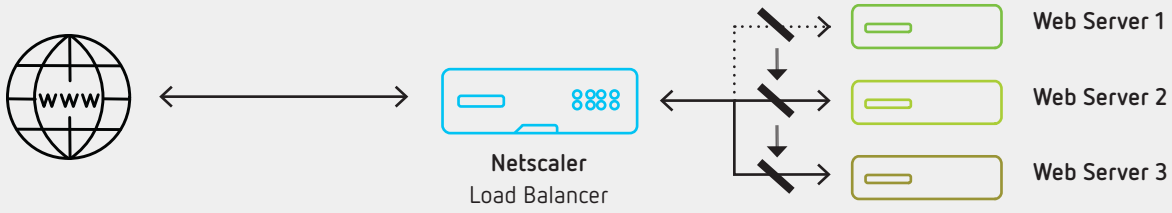
Example #2: Rolling Upgrade of a Service

A key e-commerce application has been designed using a micro services architecture. Several of the services are updated daily or weekly, or even more frequently.

These services need to be updated on several backend servers, while maintaining 100 percent availability for customers.

NetScaler Load Balancer

To manage a rolling upgrade, a NetScaler appliance suspends traffic to each server in turn while it is updated.



Let's see how Ansible can be used to take each server offline in turn to update a service, without causing any disruption to customers. While the update made here is trivial, the same process can be used to perform series of much more complex updates.

Preparation

Ansible 2.4 and the NITRO Python SDK must be installed on the machine used to run the Ansible playbooks, and that machine must be able to communicate with the REST API ports (80/443) on the target NetScaler devices (see steps in the first scenario).

The playbook

The example playbook for this scenario performs three tasks, which can be repeated for each of the backend servers. The process:

1. Takes one backend web server out of the live serving nodes.
2. Makes an update to software on that server node.
3. Brings the server back into the live serving nodes.
4. Repeats steps 1-3 until all the server nodes are updated.

The playbook uses "pre_tasks" and "post_tasks" hooks. These are Ansible features that make it easy to take a backend server offline before the update process, and bring it back up after the update has taken place.

In this playbook the "pre_tasks" section is comprised of a single task that disables the backend server in the NetScaler load balancing setup. The "post_tasks" section is a single task that re-enables the backend server in the NetScaler load balancing setup. The "serial: 1" option ensures that Ansible applies the changes one web server node at a time. Without it, Ansible would operate on all of the web servers simultaneously – and cause a service outage for customers.
The playbook applies the changes to the hosts defined in the inventory file as members of the "webservers" group.

A sample inventory file is shown below

```
[webservers]
172.30.0.21 nsip=172.30.0.10 nitro_
user=nsroot nitro_pass=nsroot
servername=webapp1 hostip=172.30.0.21
172.30.0.22 nsip=172.30.0.10
nitro_user=nsroot nitro_pass=nsroot
servername=webapp2 hostip=172.30.0.22
```

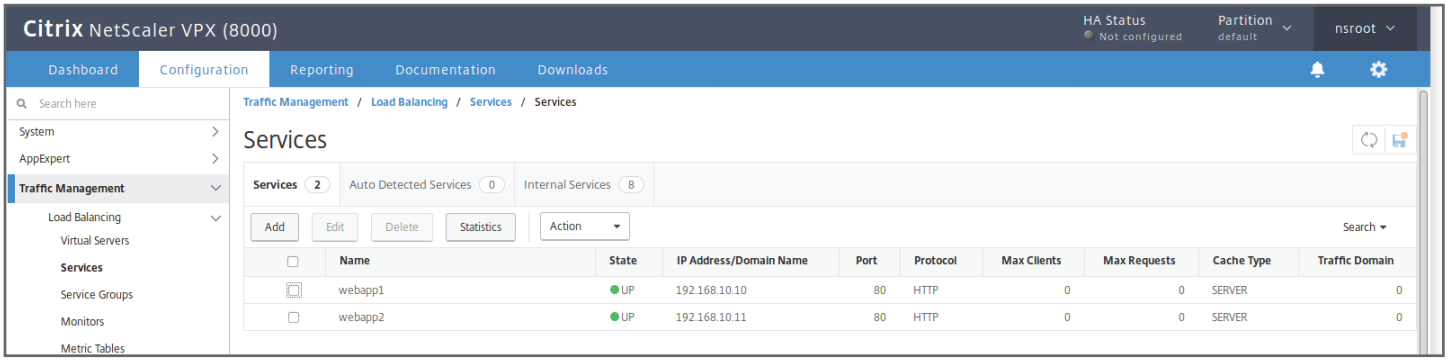
```
- hosts: webservers

  remote_user: root
  gather_facts: False
  serial: 1

  pre_tasks:
   - name: "Disable {{ servername }}"
     delegate_to: localhost
     netscaler_server:
       nsip: "{{ nsip }}"
       nitro_user: "{{ nitro_user }}"
       nitro_pass: "{{ nitro_pass }}"
       disabled: yes

       name: "{{ servername }}"
       ipaddress: "{{ hostip }}"
  post_tasks:

   - name: "Re enable {{ servername }}"
     delegate_to: localhost
     netscaler_server:
       nsip: "{{ nsip }}"
       nitro_user: "{{ nitro_user }}"
       nitro_pass: "{{ nitro_pass }}"
       name: "{{ servername }}"
       ipaddress: "{{ hostip }}"

  tasks:

   - name: "Update {{ servername }}"
     delegate_to: localhost
     command: docker-compose exec -d "{{
servername }}" bash -c "echo 'hello updated
{{ servername }}' > /app/content.txt"
```

Output

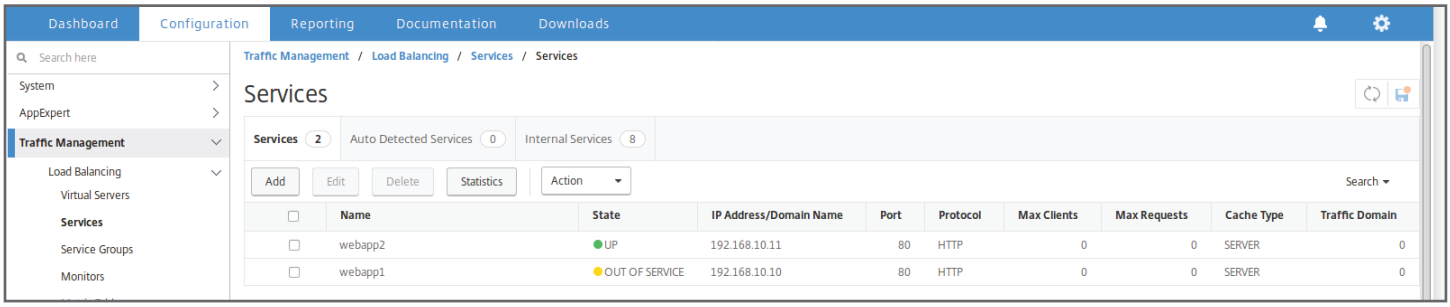
Here is the output of the playbook run. It shows that the rolling upgrade process was applied to each webserver in order. It also shows that three tasks were run for each web server (ok=3), and that one made configuration changes (changed=1). The disable and enable tasks did not apply changes.

```
PLAY [webservers] ***************************************************************************

TASK [Disable webapp1] **********************************************************************
ok: [172.30.0.21 -> localhost]

TASK [Update webapp1] ***********************************************************************
changed: [172.30.0.21 -> localhost]

TASK [Re enable webapp1] ********************************************************************
ok: [172.30.0.21 -> localhost]

PLAY [webservers] ***************************************************************************

TASK [Disable webapp2] **********************************************************************
ok: [172.30.0.22 -> localhost]

TASK [Update webapp2] ***********************************************************************
changed: [172.30.0.22 -> localhost]

TASK [Re enable webapp2] ********************************************************************
ok: [172.30.0.22 -> localhost]

PLAY RECAP ********************************************************************************
********************************
172.30.0.21              : ok=3    changed=1    unreachable=0    failed=0
172.30.0.22              : ok=3    changed=1    unreachable=0    failed=0
```

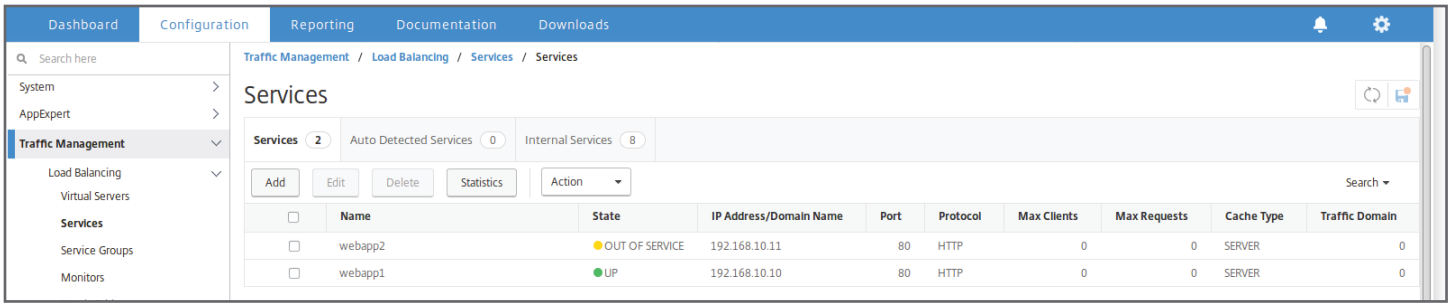
The NetScaler screen below shows that before the update both services are UP.



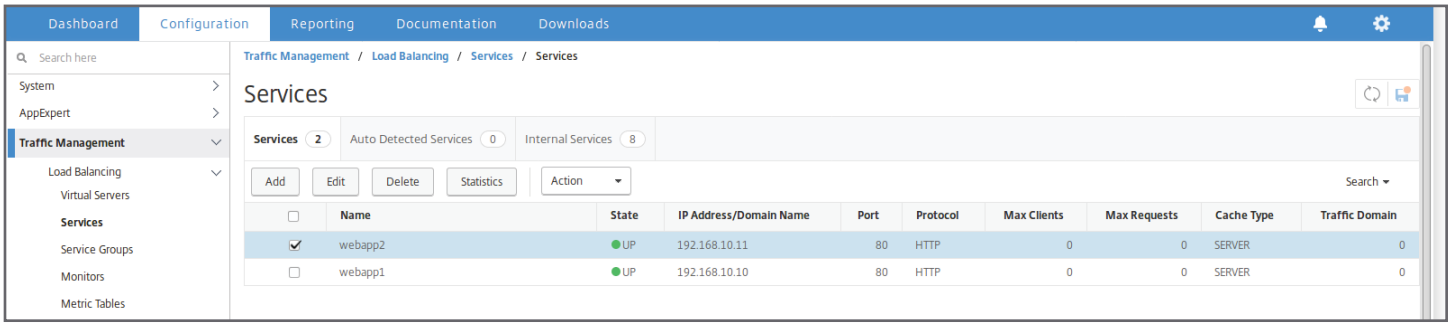
This screen shows that during the first step of the process webapp1 has been taken offline. This is the time when the backend software can be updated.



During the second step of the process, the updated webapp1 service has been brought back online, and the webapp2 service has been taken offline so it can be updated.



At the end of the process, both services are brought online and the rolling update is complete.

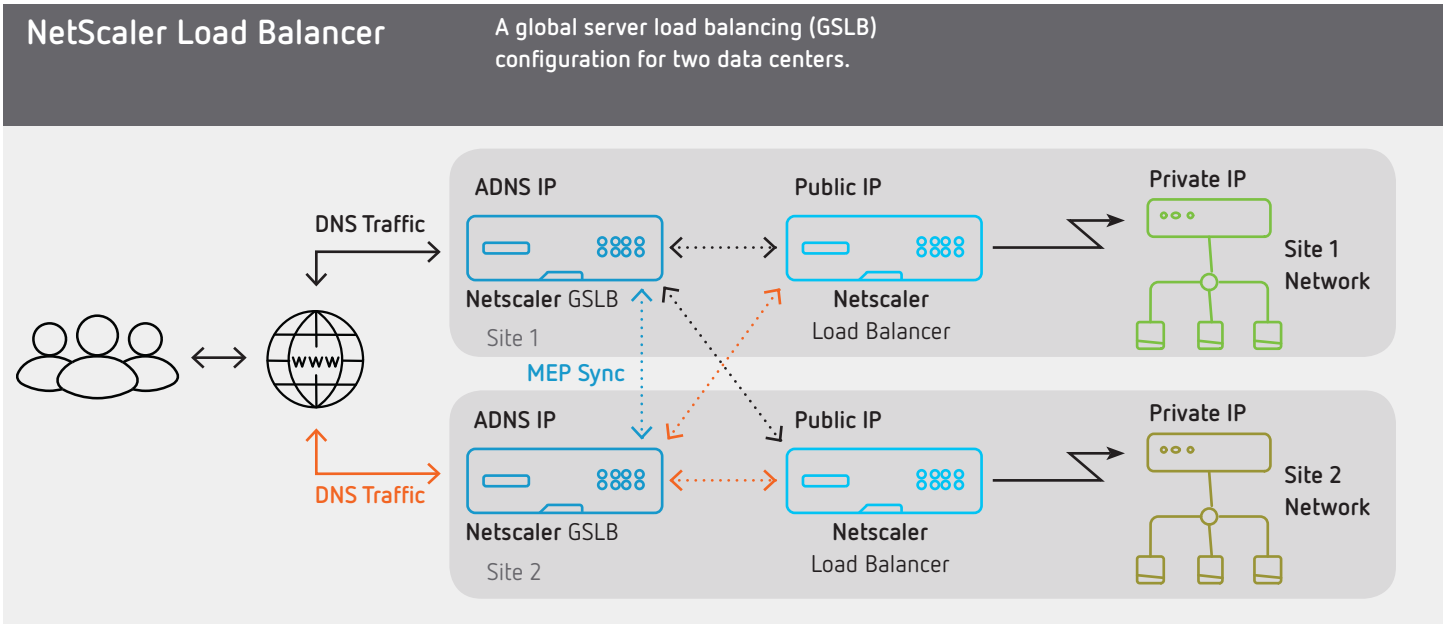


Example #3: Configuring Global Server Load Balancing

The scenario

Management decides that a mission-critical application must be continuously available, even if an entire data center fails. NetScaler appliances configured for global server load balancing (GSLB) improve application performance by directing client requests to the closest or best performing data center. In addition, if a data center experiences an outage, they can provide for disaster recovery and ensure continuous availability by shifting all traffic to surviving data centers.

For more information about global server load balancing, see the Citrix documentation web site: http://docs.citrix.com/en-us/netscaler/12/global-server-load-balancing.html



Let's see how Ansible can be used to configure NetScaler appliances to provide load balancing and high availability across data centers.

Preparation

Ansible 2.4 and the NITRO Python SDK must be installed on the machine used to run the Ansible playbooks, and that machine must be able to communicate with the REST API ports (80/443) on the target NetScaler devices (see steps in the first scenario).

The playbook

The main entities in a GSLB configuration are:

- GSLB sites represent a data center that hosts services available to the Internet. A GSLB site can be either remote or local. For high availability, two GSLB sites must be defined.
- GSLB services represent a set of services that are available either locally or remotely.
- GSLB vservers are virtual servers that load balance traffic across GSLB services.

In this playbook we create a GSLB configuration in one data center. We set up an http GSLB vserver which has two GSLB service bindings, one local and one remote. The vserver binds to the "example.com" domain name.

With this setup any requests to the "example.com" domain will be load balanced between the two sites. Additional load balancing may be performed among nodes of the local data center. High availability is accomplished by each site receiving Metric Exchange Protocol messages on the configured GSLB site IP address.

Notes:
(Required tasks not shown here)

- Setting up an Authoritative DNS server for the example.com domain name.
- Configuring a content switching or load balance virtual server for the local service IP address.
- Configuring a second GSLB site with the GSLB site types reversed, to enable a complete high availability setup.

Here is the playbook.

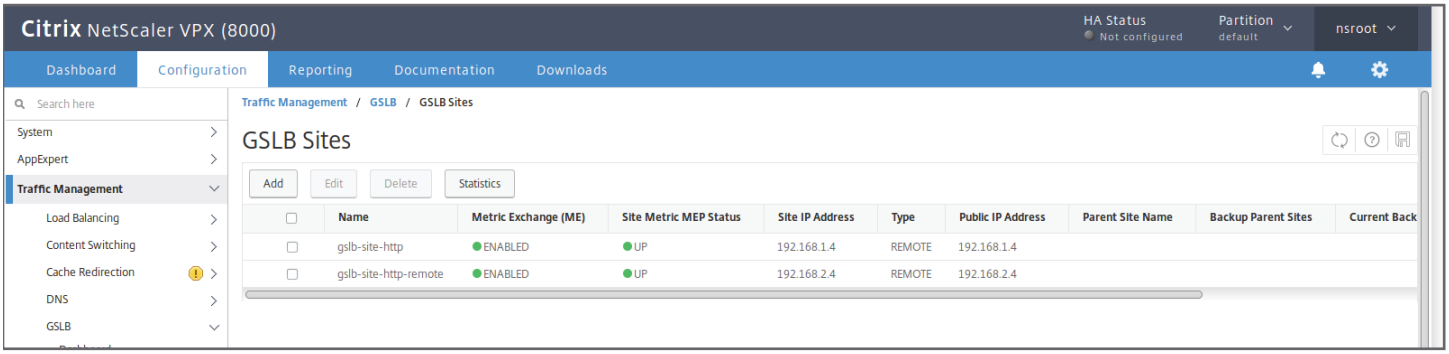
```
gather_facts: no

 tasks:

  - name: Setup local gslb site
    delegate_to: localhost
    netscaler_gslb_site:
      nsip: "{{ nsip }}"
      nitro_user: "{{ nitro_user }}"
      nitro_pass: "{{ nitro_pass }}"

      sitename: gslb-site-http
      siteipaddress: 192.168.1.4

  - name: Setup remote gslb site
    delegate_to: localhost
    netscaler_gslb_site:
      nsip: "{{ nsip }}"
      nitro_user: "{{ nitro_user }}"
      nitro_pass: "{{ nitro_pass }}"

      sitetype: REMOTE
      sitename: gslb-site-http-remote
      siteipaddress: 192.168.2.4
  - name: Setup local gslb service
    delegate_to: localhost
    netscaler_gslb_service:
      nsip: "{{ nsip }}"
      nitro_user: "{{ nitro_user }}"
      nitro_pass: "{{ nitro_pass }}"

      state: present

      servicename: gslb-service-http
      servicetype: HTTP
      port: 80
      ipaddress: 10.0.0.1
      sitename: gslb-site-http

  - name: Setup remote gslb service
    delegate_to: localhost
    netscaler_gslb_service:
      nsip: "{{ nsip }}"
      nitro_user: "{{ nitro_user }}"
      nitro_pass: "{{ nitro_pass }}"

      state: present

      servicename: gslb-service-http-remote
      servicetype: HTTP
      port: 80
      ipaddress: 10.10.0.1
      sitename: gslb-site-http-remote

  - name: Setup gslb vserver
    delegate_to: localhost
    netscaler_gslb_vserver:
      nsip: "{{ nsip }}"
      nitro_user: "{{ nitro_user }}"
      nitro_pass: "{{ nitro_pass }}"

      state: present

      name: gslb-vserver-http
      servicetype: HTTP

      domain_bindings:
        -
          domainname: example.com
          ttl: 400

      service_bindings:
        - servicename: gslb-service-http
          weight: 50
        - servicename: gslb-service-http-
remote
          weight: 50
```

Output

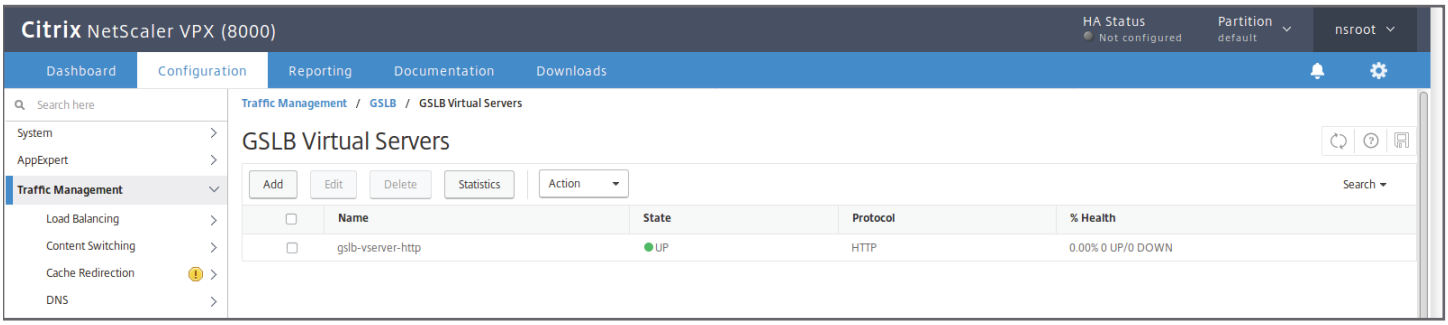
Here is the output of the playbook run.

```
PLAY [netscaler] ***************************************************************************************************************************

TASK [Setup local gslb site] ***************************************************************************************************************
changed: [netscaler_vpx120 -> localhost]

TASK [Setup remote gslb site] **************************************************************************************************************
changed: [netscaler_vpx120 -> localhost]

TASK [Setup local gslb service] ************************************************************************************************************
changed: [netscaler_vpx120 -> localhost]

TASK [Setup remote gslb service] ***********************************************************************************************************
changed: [netscaler_vpx120 -> localhost]

TASK [Setup gslb vserver] ******************************************************************************************************************
changed: [netscaler_vpx120 -> localhost]

PLAY RECAP *********************************************************************************************************************************
netscaler_vpx120           : ok=5    changed=5    unreachable=0    failed=0
```

The NetScaler screen below shows that both the local and remote GSBL sites are live.



This screen shows that the GSLB vserver is now a public-facing HTTP server and will load balance traffic between the two sites.



Conclusion

Ansible is a powerful tool for automating the deployment and management of NetScaler appliances.

Ansible and NetScaler together can help application development teams stage, deploy, and update complex, multi-tier applications quickly and reliably, without disrupting ongoing business processes. They can help network administrators and IT operations groups reduce the cost of and risk of making hundreds of changes to the network. They can help everyone deliver more value to their organizations with fewer resources.

We hope this white paper also showed you how easy it is to understand and create Ansible playbooks for NetScaler, so you can take advantage of these capabilities.

For more information on automation and how Citrix NetScaler and Ansible partnership can help your organization, visit https://www.ansible.com/ansible-netscaler.