Introduction

NetScaler Next-Gen API is a powerful modern RESTful API that allows you to programmatically configure NetScaler in a simple and intuitive way. It is based on a declarative, desired state and application-centric interface, and aims to abstract away and simplify many of the low-level complexity of traditional NetScaler configurations, making it more suitable to application developers even those who are not networking or ADC experts.

The core concept in the Next-Gen API is the *application*. All the configuration elements pertaining to the same application are grouped together, making it easy and safe to apply changes atomically.

To illustrate the new API, let's start with a simple configuration use case of creating a NetScaler load-balancing configuration with a VIP and a couple of backend servers

```
POST /mgmt/api/nextgen/v1/applications



{

  "application": {

    "name": "app1",

    "virtual_ip": "70.122.23.11",

    "default_certificate_ref": "app1_cert",

    "servers": [

      "192.168.10.11",

      "192.168.10.14"

    ]

  }
```

```
}
```

With this one API call, we have fully configured this load-balancing use case. We'll cover how we got a certificate reference later in the document.

Let's say we want to update the configuration to add more servers or make other changes. We can simply issue another API call to add more servers as follows:

```
PUT /mgmt/api/nextgen/v1/applications/app1



{

  "application": {

    "name": "app1",

    "virtual_ip": "70.122.23.11",

    "default_certificate_ref": "app1_cert",

    "servers": [

      "192.168.10.11",

      "192.168.10.14",

      "192.168.10.15",

      "192.168.10.16"

    ]

  }

}
```

Notice that we have provided the new "desired state" of the application in the update API, which now contains 4 servers rather than 2.

Subsequent sections of this documentation cover advanced use case scenarios, which includes content switching, configurations involving multiple VIP addresses, and diverse port setups.

Comparison to Nitro API

While the NetScaler traditional Nitro API has been very successful in providing a REST API for interfacing programmatically with NetScaler and unlocking all its features, it ultimately is an API that mirrors the internal NetScaler processing constructs, the NetScaler CLI structure, and reflects the low-level complexity and idiosyncrasies of the NetScaler configuration model making its usage intimidating to many new users.

The Next-Gen API takes a new approach of a user-centric design, starting from the common use cases and patterns that customers often try to realize on NetScaler, and making these use cases explicit and very simple to create. It also aims to be developer-friendly by adopting consistency in behavior and conventions, a logical structure, with modern desired state semantics.

Using the Nitro API, the simple load-balancing configuration shown above, would require 6 API calls, and assumes acquaintance with concepts and APIs for creating vservers, servicegroups, services and bindings between these entities. Slightly more complex use cases can require tens of API calls, with the user or the calling application responsible for understanding the proper order of calling these APIs and handling any mid-workflow errors.

The other difference with the Nitro API has to do with the processing infrastructure. The Next-Gen API uses a new modern config infrastructure on NetScaler that segregates the config validation and commit stage from applying the config to the data plane. What this means for users is faster API processing. The config is considered "committed" once it's validated and stored on a persistent DB. In the background and asynchronously, the Next-Gen config infrastructure will translate and apply the latest config changes from the DB to the NetScaler data plane (packet engines).

Configuration Views

Both the Next-Gen API and Nitro API are supported on NetScaler, and configurations can be created using both. Configurations created with Next-Gen API can only be modified using Next-Gen API. However, these configurations can be viewed in read-only mode through the CLI or Nitro API. To switch to the Next-Gen API config view, use the following CLI command:

```
switch ns configview NEXTGENAPI
```

In this view, you can only see configurations created through Next-Gen API. By default, the config view is set to `ALL`, which provides a read-only view of Next-Gen API configurations along with an unrestricted view of configurations created using NetScaler CLI, GUI, or Nitro API. Additionally, there is the `CLASSIC` config view, where you can view and modify configurations created using the NetScaler CLI, GUI, or Nitro API.

> **Note:**

> Configview is a session-level setting that applies only to the current user session (CLI or Nitro session) and does not affect all users globally.

> Configuration created through Next-Gen API is not saved in `ns.conf` when you run the `save ns config` command. Next-Gen API configuration is persistent and remains intact after a reboot.

The following behavior is observed for different levels of the `clear ns config <level>` command:

LEVEL	DESCRIPTION
basic	Configuration created through Next-Gen API is not cleared.
extended	Configuration created through Next-Gen API is not cleared.
full	Configuration created through Next-Gen API is cleared.

To delete all Next-Gen API resources at once, use the following CLI command:

```
clear ns nextgenapi
```

To delete individual resources like applications or certificates, use the respective Next-Gen DELETE APIs.

System Requirements

Next-Gen API is available on NetScaler release 14.1 12.30.

If you are using NetScaler VPX, please ensure you have at least 4GB RAM.

> **Note:**
>
> Currently, NetScaler standalone and HA deployment is supported on MPX and VPX. Cluster is not yet supported.

Enabling Next-Gen API on NetScaler

Before using Next-Gen API, you need to enable it first on NetScaler using the CLI:

```
> enable ns nextgenapi



ERROR: Operation not permitted [To use nextgenapi you need to enable SSL
defaultprofile. CLI CMD: set ssl parameter -defaultProfile ENABLED]
```

This will fail if you don't already have defaultProfile enabled for SSL as Next-Gen API requires this. Once you configure the default profile for SSL, retry this command.

You can verify the status of Next-Gen API using this command:

```
> show ns nextgenapi



Next-Gen API State: STARTED



>
```

Now, you can start exploring its new APIs.

API reference

Next-Gen API reference documentation is specified using the OpenAPI specification. The same specification used for documentation is used to generate API server stubs to ensure that the documentation always reflects the actual specification.

You will also find multiple examples of each API endpoint in the reference documentation to illustrate usage. You can browse the API documentation on the NetScaler GUI under the **Documentation** tab, and download a copy of the YAML OpenAPI specification from the **Downloads** tab.

Authentication

Next-Gen API requires users to first authenticate using a valid user account on NetScaler in order to obtain a session token. Once authenticated, the returned cookie needs to be included in all subsequent API calls in the form of a cookie.

Use the following API for authentication and obtaining a session token.

```
POST /mgmt/api/nextgen/v1/login



{

  "login": {

    "username": "user1",

    "password": "verysecret"

  }

}
```

If authentication is successful, the returned status code is 200, and the response will include a Set-Cookie header that will set a cookie named **sessionid** which contains a valid session token to be used for subsequent API calls.

```
200 OK

Set-Cookie: sessionid=%23%23B2A100C6D1F36AAE2......A50FA2C576171;

Path=/mgmt/api/nextgen/v1
```

You can also specify the duration of the created session, by adding a timeout attribute to the login request as shown below:

```
POST /mgmt/api/nextgen/v1/login



{

  "login": {

    "username": "user1",

    "password": "verysecret",

    "timeout": "5min"

  }

}
```

In the above example, we are requesting a session that will timeout after 5 minutes of inactivity.

Note:

In Next-Gen API, all values that represent time duration, will always have the unit of time as part of the value. For example, a 5 minutes value, can be given as **5m**, **5min**, **300s** or **300sec**.

After a successful login, subsequent API requests should include the sessionid cookie header. For example, to list all applications, the following API call with the sessionid cookie is made:

```
GET /mgmt/api/nextgen/v1/applications



Cookie: sessionid=%23%23B2A100C6D1F36AAE2......A50FA2C576171
```

Next-Gen API also provides an explicit call to logout from the session, which invalidates the session token.

```
POST /mgmt/api/nextgen/v1/logout



Cookie: sessionid=%23%23B2A100C6D1F36AAE2......A50FA2C576171
```

The following is a successful response to the logout API call:

```
200 OK



Set-Cookie: sessionid=deleted; expires=Thu, 01-Jan-1970 00:00:01 GMT; Max-
Age=0; Path=/mgmt/api/nextgen/v1
```

The response doesn't contain a body payload. Once successfully logged out, the session cookie is no longer valid.

Note:

Next-Gen API and Nitro API share the same system users. You can configure users who can access the appliance using regular NetScaler CLI or GUI.

Role-Based Access

Next-Gen API allows you to define the set of API endpoints that a user can call through the ability of defining roles, and associating allowed operations and API endpoints with each role. You can then assign these roles to user groups.

The following API creates a role called app-admins that allows users all operations on the applications resources.

```
POST mgmt/api/nextgen/v1/roles/apps-admin-role



{

  "role": {

    "name": "apps-admin-role",

    "permissions": [

      {

        "method": "ALL",

        "resources": "/applications/.+"

      }

    ]

  }

}
```

The role has a name and a set of permissions. Each permission specifies the resource URLs allowed (using a regular expression) along with the HTTP methods on these resources. In the example above, the role allows all HTTP methods (GET, POST, PUT, DELETE) on any application by specifying the regular expression *"/applications/.+"*.

Once the role is defined, it can be assigned to one or multiple user groups. The following API assigns the role "apps-admin-role" to the user group "apps-admin-grp".

```
POST mgmt/api/nextgen/v1/groups/apps-admin-grp/roles
```

```
{

  "role": "apps-admin-role"

}
```

Once a role is assigned to a group, it cannot be deleted until it is dissociated from any user groups where it is being used. You can remove the association between a user group and a role using the following API

```
DELETE mgmt/api/nextgen/v1/groups/apps-admin-grp/roles/apps-admin-role
```

Now, when a user logs in through Next-Gen API, their user groups are retrieved, and based on the roles for these groups, the logged in user will be restricted to the API endpoints allowed by these roles.

If a user tries to access an API endpoint for which they are not authorized, they will receive a 401 Not Authorized error.

Note:

Configuring user accounts, user groups and managing user authentication on NetScaler, including configuring external authentication such as LDAP or RADIUS can continue to be managed using NetScaler CLI, GUI, or Nitro API. Please refer to NetScaler documentation for these topics.

Configuration Paradigms

NetScaler Next-Gen API supports two complementary configuration paradigms designed to address different use cases and user requirements:

- **Application-Centric Model**: The application-centric approach groups all configuration elements related to an application together, making it easy to manage complete application configurations atomically. This model is ideal for common use cases and provides a simplified, high-level abstraction.

For detailed API reference information about the Application-Centric model, including resource types, API structure, and semantics, see the Application-Centric API Reference.

- **Entity-Centric Model (Config Sets)**: For scenarios requiring full access to NetScaler's feature set or granular control over individual Nitro entities, Config Sets provide a declarative mechanism to define any NetScaler configuration. This bridges the gap between the power of Nitro and the simplicity of Next-Gen API's desired-state model.

For detailed information about Config Sets, including use cases, examples, and best practices, see the Config Sets Guide.

API Calls Structure

Next-Gen API resources and calls all follow a consistent format.

```
<http_verb> /mgmt/api/nextgen/v1/<resource_type>/<resource_name>
```

For example to retrieve the details of an application called my_app:

```
GET /mgmt/api/nextgen/v1/applications/my_app
```

If you want to retrieve all applications, you omit the resource_name part:

```
GET /mgmt/api/nextgen/v1/applications
```

Some resources are sub-resources of existing resources, in which case, the URL follows this format:

```
<http_verb> /mgmt/api/nextgen/v1/<resource_type>/<resource_name>/
<subresource_type>/<subresource_name>
```

For example to retrieve the backends of an application called my_app, you use the following API call:

```
GET /mgmt/api/nextgen/v1/applications/my_app/backends
```

To retrieve a specific backend named metadata_servers, you use the following call:

```
GET /mgmt/api/nextgen/v1/applications/my_app/backends/metadata_servers
```

The supported HTTP verbs are the following:

VERB	DESCRIPTION
GET	This verb is used to retrieve a specific resource by name or a list of resources for a certain type.
POST	This verb is used to create or update the "desired state" of a resource. If a resource is successfully created, the returned status code is 201. If an existing resource was updated, the returned status code is 200.
PUT	This verb is used to update the "desired state" of an existing resource.
DELETE	This verb is used to delete an existing resource.

The request and response payloads are expressed in JSON format.

Resource Types and Resources

Next-Gen API config model is made up of a set of resource types. The following are the resource types currently supported:

- Application
  - Backend
    - Server
  - Frontend
    - Listener
  - Route
- Certificate

- Filter Value Set
- Responder HTMLPage
- HTTP Callouts
- Config Sets

Resource types are organized in a hierarchy: some resource types are subtypes of others as indicated by the indentation above. For example, frontends and backends are subtypes of the application resource type. And servers is a subtype of a backend. This hierarchy is reflected in the URL representation of a resource.

```
/mgmt/api/nextgen/v1/applications/{application_name}/backends/
{backend_name}/servers
```

Here the URL reflects that servers is a subresource of a specific backend, which is a subresource of a specific application.

The following table describes the role of each resource supported in the API:

RESOURCE	PARENT RESOURCE	DESCRIPTION
Application	N/A	The application resource is the top resource that represents a complete configuration.
Frontend	Application	An application can have multiple frontends. Each frontend represents a virtual IP. For example, an application can have an internal virtual IP and an external virtual IP. Most applications have only one virtual IP, in which case the frontend is implicit.
Listener	Frontend	A Listener represents a port on the virtual IP. Some applications can listen on multiple ports (for example, HTTP/80 and HTTPS/443)

RESOURCE	PARENT RESOURCE	DESCRIPTION
		and therefore need multiple listeners. Often, applications listen only on one port, in which case the listener is implicit.
Backend	Application	A Backend represent a set of servers that serve the same traffic and for which incoming client requests can be load-balanced between them.
Servers	Backend	The list of server IPs/Ports that make up a backend.
Certificate	N/A	A certificate is a top resource. A certificate represents a public/private key pair and any certificate chain (intermediate and root certificates).

Listeners

The listener has the following attributes:

ATTRIBUTE NAME	ATTRIBUTE TYPE	DESCRIPTION	REQUIRED?
name	name	The name of the listener.	Yes
port	port	The port number of the listener	Yes
protocol	protocol	Protocol for this port. If not specified, defaults to HTTP	No

ATTRIBUTE NAME	ATTRIBUTE TYPE	DESCRIPTION	REQUIRED?
default_certificate_ref	reference	The name of a server certificate to be used on this port.	Depends. If protocol is HTTPS, a certificate must be specified.
certificate_refs	List of references	The names of a server SNI certificates to be used on this port.	If protocol is HTTPS. Can be used with default_certificate_ref.

Applications

In summary, a simple LB application can have the following attributes:

ATTRIBUTE NAME	ATTRIBUTE TYPE	DESCRIPTION	REQUIRED?
virtual_ip	ipaddress	The VIP of the application	Yes
port	port	The port number of this VIP. Defaults to either 443 or 80 based on the protocol.	No
protocol	protocol	The protocol of the VIP. Defaults to HTTPS if not specified	No
default_certificate_ref	reference	The name of a server certificate to be used on this port.	Depends. If protocol is HTTPS, a certificate must be specified.
certificate_refs	List of references	The names of a server SNI certificates to be used on this port.	If protocol is HTTPS. Can be used with default_certificate_ref.

ATTRIBUTE NAME	ATTRIBUTE TYPE	DESCRIPTION	REQUIRED?
servers_port	port	The port number for the backend servers. Defaults to 80	No
servers_protocol	protocol	The protocol of the servers. Defaults to HTTP if protocol is HTTP or HTTPS.	No
servers	server list	A list of backend servers that will server requests of this application	Yes

References between resources

The resource's name specified during creation can be used to refer to it from other resources. We used this already to refer to a certificate from an application payload.

Next-Gen API uses the naming convention `<resource_type>_ref` for attribute names that carry a reference to an existing resource. For plural references, the naming convention is `<resource_type>_refs`. For example, the attribute **certificate_refs** of an application resource.

Desired State Semantics

Next-Gen API allows users to specify the "desired state" of a resource, and let the system figure out how to bring the configuration from its "current state" to that "desired state".

For example, when the user wants to ensure that an application called "app1" with a certain configuration is present on NetScaler, they can simply issue the following API call:

```
POST /mgmt/api/nextgen/v1/applications/app1



{

  "application": {
```

```
    "name": "app1",
    "virtual_ip": "70.122.23.11",
    "protocol": "HTTP",
    "servers": [
      "192.168.10.11",
      "192.168.10.14"
    ]
  }
}
```

It doesn't matter whether application "app1" already exists or whether it's newly created. Next-Gen API will ensure to create it if it doesn't exit or to update it with the specified configuration if it already exists.

Another example would be configuring the backend servers of an application.

```
PUT /mgmt/api/nextgen/v1/applications/app1/backends/tier1/servers


{
  "servers": [
    "192.168.10.11",
    "192.168.10.14",
    "192.168.10.15",
    "192.168.10.16",
    "192.168.10.17"
```

```
  ]

}
```

It doesn't matter whether there are servers already in the backend called "tier1" or not. This API will ensure that after it's executed the backend "tier1" will have the specified servers in the request. If some of these servers are already present, then only the missing servers are added. If there are servers that are not specified in the request, then these servers will be removed from the backend.

The "desired state" semantics frees the user from having to track the current state of resources, and having to imperatively specify the steps required to bring the resource to the user's desired state.

Incremental Change Semantics

While "desired state" is very useful for simplifying the API client, it is also desirable in some cases to avoid specifying the whole resource state on each API call, and only specify what needs to be changed (added or removed). Taking the previous example, if the backend "tier1" has a 1,000 servers currently, and we want to add 1 extra server, it's bulky to specify 1,001 servers in the request, and would be more efficient to simply specify the new server to be added. Next-Gen API also accomodates this usage pattern by providing an explicit API call to change a resource's state explicitly.

For example, to add a new server to the backend "tier1" (without respecifying all the servers that are already part of this backend), we can use the following API call:

```
POST /mgmt/api/nextgen/v1/applications/app1/backends/tier1/servers



{

   "server": "192.168.10.17"

}
```

Similarly, if we want to create a new application called "app1", and we want the operation to fail if "app1" already exists, we can use the following POST API:

```
POST /mgmt/api/nextgen/v1/applications



{

 "application": {

    "name": "app1",

    "virtual_ip": "70.122.23.11",

    "protocol": "HTTP",

    "servers": [

      "192.168.10.11",

      "192.168.10.14"

    ]

  }

}
```

The difference with the "desired state" API described in the previous section is a subtle one: we ommitted the application name "app1" from the URL.

```
POST /mgmt/api/nextgen/v1/applications/app1
```

Application actions

The following actions are supported for the application:

Disable/Enable Application

To disable the application so it doesn't serve any traffic, use the following API call:

```
POST /mgmt/api/nextgen/v1/applications/{application-name}/actions/disable
```

To enable the application so it can serve traffic, use the following API call:

```
POST /mgmt/api/nextgen/v1/applications/{application-name}/actions/enable
```

The enable/disable application APIs are reflected in the `configured_state` parameter in the application health response.

> **Note:**
>
> Disable/Enable APIs are also supported for application subcomponents such as frontends, listeners, backends, and servers. For more information, see API Exploration section of the document.

Uninstall/Install Application

To uninstall the application, meaning the application config is removed from NetScaler data plane(packet engines) but remains on Next-Gen. , use the following API call:

```
POST /mgmt/api/nextgen/v1/applications/{application-name}/actions/
uninstall
```

This operation is typically useful when there are discrepancies observed in data plane behavior due to config inconsistencies, and you want to clear the config from the data plane and reinstall it later.

To install the uninstalled application, use the following API call:

```
POST /mgmt/api/nextgen/v1/applications/{application-name}/actions/install
```

Error Handling

Next-Gen API uses HTTP Status Codes to signal success or failure. Successful API responses are identified by the HTTP status codes 200 or 201.

Below are the list of HTTP Status codes that can be returned by the API and their meaning:

HTTP STATUS CODE	DESCRIPTION
200	This status code indicates that the operation is successful. This can be returned for GET, POST and PUT API calls.
201	This status code indicates that the resource creation is successful. This can be returned for POST API calls.
400	This status code indicates an error in the request. The response body will contain details about the type of the error.
401	This status code indicates a failure in authenticating the user. This could also be because of the existing user session has expired and the user needs to logon again. The response contains details about the error.
403	This status code indicates that the user does not have permission to make this API call on the resource referred to in the URL.
404	This status code indicates the referred resource in the GET, POST, PUT or DELETE URL doesn't exist.
409	This status code indicates a failure because of a conflict. For example, if a user tries to create a resource which already exists on NetScaler, this status code is used to indicate the conflict. The response body will contain details of the error.
500	This status code indicates an internal error on NetScaler. The response body contains details onto the nature of the error. User needs to troubleshoot further and return NetScaler to a working condition.

When the HTTP status code indicates an error (400-500 range), the response body contains the details of the error. For example, below is the response body for a 400 error returned, indicating the request payload is missing a required attribute (certificate reference).


```
{

    "errorcode": 400,

    "errormessage": "Validation Error",

    "details": [

        {

            "instance": "{'name': 'color_app', 'virtual_ip':
'10.102.201.171', 'protocol': 'HTTPS', 'service_port': 8010, 'servers':
['10.102.201.166', '10.102.201.167']}",

            "message": "'certificate_ref' is a required property for
'protocol':'HTTPS'"

        }

    ]

}
```


The following table enumerates the mapping between HTTP Status codes and the internal error codes and error messages:

HTTP STATUS CODE	INTERNAL ERROR CODE	INTERNAL ERROR MESSAGE
400	1000	Bad Request
400	1001	Invalid JSON data
400	1002	Resource name is different in URL and payload
400	1003	Missing query parameters

HTTP STATUS CODE	INTERNAL ERROR CODE	INTERNAL ERROR MESSAGE
401	1100	Unauthorized Request
401	1101	Session expired or killed. Please login again
401	1102	User not authorized for this operation
401	1103	Authentication failed - Please provide sessionid in the cookie
401	1104	Invalid username or password
401	1105	Operation not permitted on HA secondary node
404	1200	Resource not found
404	1201	Application not found
404	1202	Frontend not found
404	1203	Listener not found
404	1204	Backend not found
404	1205	Server not found
404	1206	Route not found
404	1207	Certificate not found
404	1208	Role not found
404	1209	Roles not found
404	1210	No Role associations found for group

HTTP STATUS CODE	INTERNAL ERROR CODE	INTERNAL ERROR MESSAGE
404	1211	Value set not found
404	1212	Application not installed
404	1213	Responder HTML page not found
404	1214	HTTP callout not found
406	1300	We only support sending responses in JSON format
409	1400	Resource already exists
409	1401	Delete of resource already in progress
409	1402	Delete of this resource is not possible as it is being referenced by other resources
409	1403	Uninstall of this resource is not possible as it is being referenced by other resources
409	1410	Application already exists
409	1411	Frontend already exists
409	1412	Listener already exists
409	1413	Backend already exists
409	1414	Server already exists
409	1415	Route already exists
409	1416	Certificate already exists

HTTP STATUS CODE	INTERNAL ERROR CODE	INTERNAL ERROR MESSAGE
409	1417	Role already exists
409	1418	Group Role association already exists
409	1419	Value set already exists
409	1420	Blocking operation is in Progress, please try after sometime
409	1421	Responder HTML page already exists
409	1422	HTTP callout already exists
415	1500	We only support requests in JSON format
422	2000	Validation Failed
422	3001	The resource cannot be deleted because it is the last remaining resource of its kind
500	1600	Internal Server Error

Note:

Please refer to the OpenAPI spec for an up-to-date information on all the valid errorcode values that can be returned by the API.

Application-Centric API Reference

This document provides detailed reference information for the Application-Centric API model in NetScaler Next-Gen API.

Application Types

Next-Gen API allows users to configure different types of applications from the simple applications that consist of one VIP/port to more sophisticated applications that feature multiple VIPs/Ports and multiple backend server groups.

Simple Applications

For a simple application listening on one virtual IP and one port, you can create one with the following sample JSON payload:

```
{
  "application": {
    "name": "my_app",
    "virtual_ip": "70.122.23.11",
    "port": 443,
    "default_certificate_ref": "cert1",
    "servers_port": 8080,
    "servers": [
      "192.168.10.11",
      "192.168.10.14"
    ]
```

```
  }
}
```

The payload directly specifies the virtual IP and port as well as the backend servers and their port. Notice that we didn't specify the VIP's protocol. In this case, it is assumed to be the default protocol "HTTPS". Actually, even the VIP's port number is optional in this case: If not specified, it defaults to the protocol's default port (for HTTPS, it is 443). So we could rewrite our payload simply as:

```
{
  "application": {
    "name": "my_app",
    "virtual_ip": "70.122.23.11",
    "default_certificate_ref": "cert1",
    "servers_port": 8080,
    "servers": [
      "192.168.10.11",
      "192.168.10.14"
    ]
  }
}
```

You don't have to use the compact notation "server_ip". If you want to be clear and explicit, you can use the following expanded format for specifying each server's ip and port through explicit attributes:

```
{
  "application": {
```

```
    "name": "my_app",
    "virtual_ip": "70.122.23.11",
    "port": 443,
    "default_certificate_ref": "cert1",
    "servers_port": 8080,
    "servers": [
       {
          "ip": "192.168.10.11"
       },
       {
          "ip": "192.168.10.14"
       },
       {
          "ip": "192.168.10.15"
       },
       {
          "ip": "192.168.10.16",
          "port": 8181
       }
    ]
  }
}
```

You can also mix both styles when specifying servers as shown in the example below:

```
{
  "application": {
    "name": "my_app",
    "virtual_ip": "70.122.23.11",
    "port": 443,
    "default_certificate_ref": "cert1",
    "servers_port": 8080,
    "servers": [
        "192.168.10.11",
        "192.168.10.14",
        "192.168.10.15",
        {
          "ip": "192.168.10.16",
          "port": 8181
        }
    ]
  }
 }
}
```

Notice that the last server, we specified a port attribute that overrides the default **servers_port** attribute value.

Which notation to use (compact vs expanded) is really a matter of style preference.

You can also specify the weight of a server, if you want a server to serve more requests than other servers by a certain factor. For example, in the example below, the first and the last server serve twice as many requests as the remaining servers:

```
"servers": [
       {
          "ip": "192.168.10.11",
          "weight": 2
       },
       {
          "ip": "192.168.10.14"
       },
       {
          "ip": "192.168.10.15"
       },
       {
          "ip": "192.168.10.16",
          "port": 8181,
          "weight": 2
       }
    ]
```

Applications with one VIP and multiple ports

For applications that feature a VIP with multiple ports, we introduce the concept of a *listener*. Each port is modeled as a separate listener, as shown in the example below for an application that is listening on an HTTPS port and an HTTP port.

```
{
  "application": {
    "name": "app1",
    "virtual_ip": "70.122.23.11",
    "listeners": [
      {
        "name": "secure_listener",
        "port": 443,
        "protocol": "HTTPS",
        "default_certificate_ref": "app1_cert",
      },
      {
        "name": "http_listener",
        "port": 80,
        "protocol": "HTTP"
      }
    ],
    "servers_port": 8080,
```

```
    "servers": [
      "192.168.10.11",
      "192.168.10.14",
      "192.168.10.15",
      "192.168.10.16",
      "192.168.10.17",
      "192.168.154.47"
    ]
  }
}
```

Any requests received from either listener (HTTP/80 or HTTPS/443) will be load-balanced amongst the servers of this application.

You can also specify that any request received on the HTTP/80 port be redirected to the HTTPS port by setting the attribute http_to_https_redirect to true on the HTTPS listener.

```
{
  "application": {
    "name": "app1",
    "virtual_ip": "70.122.23.11",
    "listeners": [
      {
        "name": "secure_listener",
        "port": 443,
```

```
        "protocol": "HTTPS",
        "default_certificate_ref": "app1_cert",
        "http_to_https_redirect": true
      }
    ],
    "servers_port": 8080,
    "servers": [
      "192.168.10.11",
      "192.168.10.14",
      "192.168.10.15",
      "192.168.10.16",
      "192.168.10.17",
      "192.168.154.47"
    ]
  }
}
```

Since we only have one listener, we can also write the above payload more simply as:

```
{
  "application": {
    "name": "app1",
    "virtual_ip": "70.122.23.11",
```

```
    "protocol": "HTTPS",
    "port": 443,
    "default_certificate_ref": "app1_cert",
    "http_to_https_redirect": true,
    "servers_port": 8080,
    "servers": [
      "192.168.10.11",
      "192.168.10.14",
      "192.168.10.15",
      "192.168.10.16",
      "192.168.10.17",
      "192.168.154.47"
    ]
  }
}
```

Applications with multiple VIPs

Some applications have more than one virtual IP (VIP), maybe an external VIP and an internal VIP, but users want to view all these VIPs as part of the same application. To model each VIP in such applications, we introduce the concept of a *frontend*. Each frontend represents one VIP of the application. The frontend itself can have multiple listeners if there are multiple ports on that VIP as we have discussed in the previous section.

The following example shows an application with 2 frontends, the second frontend having an explicit listener. All requests to either frontend (VIP/port) are load-balanced amongst, and served by, the same set of backend servers.

```
{
  "application": {
    "name": "app1",
    "frontends": [
      {
        "name": "admin_frontend",
        "virtual_ip": "70.122.23.26",
        "port": 443,
        "protocol": "HTTPS",
        "default_certificate_ref": "admin_cert"
      },
      {
        "name": "customer_frontend",
        "virtual_ip": "70.122.24.21",
        "listeners": [
          {
            "name": "secure_listener",
            "protocol": "HTTPS",
            "port": 443,
            "default_certificate_ref": "app6_cert",
            "http_to_https_redirect": true
          }
```

```
        ]
      }
    ],
    "servers_port": 8080,
    "servers": [
      "192.168.10.11",
      "192.168.10.14",
      "192.168.10.15"
    ]
  }
}
```

For applications that have only one VIP as we saw earlier, you can think of them as having an implicit "default" frontend.

Applications with multiple backends

For applications where you want to route or switch requests to different groups of servers based on the characteristics of the request or the client (the "content switching" use case), you will need to specify different groups of backend servers and then specify how you route or switch traffic to different groups based on characteristics of the request or the client. For this purpose, we introduce 2 new concepts: *backends* and *routes*.

The following example shows an application with 2 backends, called respectively **account_servers** and **billing_servers**. The routes define the rules that govern which requests go to which backend. In this example, we want all requests that start with `/account` to go to the **account_servers** and all the requests that start with `/checkout` go the **billing_servers** backend.

```
{
  "application": {
    "name": "app5",
    "virtual_ip": "70.122.23.11",
    "port": 443,
    "protocol": "HTTPS",
    "default_certificate_ref": "portal_cert",
    "backends": [
      {
        "name": "account_servers",
        "servers_port": 8080,
        "servers": [
          "192.168.10.11",
          "192.168.10.14"
        ]
      },
      {
        "name": "billing_servers",
        "servers_port": 8080,
        "servers": [
          {
            "ip": "192.168.11.11"
```

```
          },
          {
            "ip": "192.168.11.14"
          }
        ]
      }
    ],
    "routes": [
      {
        "name": "account_route",
        "filter": "http.request.uri startswith '/account/'",
        "backend_ref": "account_servers"
      },
      {
        "name": "checkout_route",
        "filter": "http.request.uri startswith '/checkout/'",
        "backend_ref": "billing_servers"
      }
    ]
  }
}
```

For the previous application payloads where there was no explicit backend configured and we only specified the servers, you can think of them having an implicit "default" backend.

You also notice in the above payload that the criteria for routing or switching requests to a specific backend is specified using the "filter" attribute whose value is an expression that is easy to understand. These expressions follow the **Wireshark Display Filters** expression syntax, selected because many developers are familiar with Wireshark as a tool, and this syntax is easy to understand and remember. Section *Wireshark Expressions* covers the various expressions that can be used.

If you are already familiar with traditional NetScaler policy expressions (PI), you can still use the PIXL syntax rather than the Wireshark syntax by setting the filter_format attribute to pixl as shown below:

```
{

        "name": "account_route",

        "filter": "HTTP.REQ.URL.PATH.STARTSWITH(\"/account\")",

        "filter_format": "pixl",

        "backend_ref": "account_servers"

    }
```

Applications with multiple VIPs and multiple backends

If an application requires both multiple VIPs and multiple backends then this application will have a set of frontends, a set of backends, and routes to determine how requests are routed between the frontends and backends.

In the following example, we have an application that has 2 frontends and 2 backends.

```
{

  "application": {

    "name": "app8",
```

```
"frontends": [
  {
    "name": "admin_frontend",
    "virtual_ip": "70.122.23.26",
    "port": 443,
    "protocol": "HTTPS",
    "default_certificate_ref": "admin_cert"
  },
  {
    "name": "app_frontend",
    "virtual_ip": "70.122.24.21",
    "listeners": [
      {
        "name": "secure_listener",
        "port": 443,
        "protocol": "HTTPS",
        "default_certificate_ref": "app8_cert"
      },
      {
        "name": "http_listener",
        "port": 80,
        "protocol": "HTTP"
```

```
        }
      ]
    }
  ],
  "backends": [
    {
      "name": "account_servers",
      "port": 8080,
      "servers": [
        "192.168.10.11",
        "192.168.10.14"
      ]
    },
    {
      "name": "billing_servers",
      "port": 8080,
      "servers": [
          "192.168.11.11",
          "192.168.11.14"
      ]
    }
  ],
```

```
    "routes": [
      {
        "name": "account_route",
        "filter": "http.request.uri startswith '/account/'",
        "backend_ref": "account_servers"
      },
      {
        "name": "checkout_route",
        "filter": "http.request.uri contains '/checkout/'",
        "backend_ref": "billing_servers"
      }
    ]
  }
}
```

This is similar to the previous example, except that requests received from either frontends will be routed through the same logic to the 2 backends.

In some cases however, we want to be more specific about routing such as routing requests only from a given frontend/listener. In this case, the route needs to explicitly reference the frontend/listener that it is targeting as shown below:

```
"routes": [
      {
        "name": "checkout_route",
        "frontend_ref": "app_frontend",
```

```
        "listener_ref": "secure_listener",
        "filter": "http.request.uri contains '/checkout'",
        "backend_ref": "billing_servers"
      },
      {
        "name": "ad_route",
        "frontend_ref": "app_frontend",
        "listener_ref": "http_listener",
        "filter": "http.request.uri startswith '/ad'",
        "backend_ref": "advert_servers"
      },
      {
        "name": "account_route",
        "filter": "http.request.uri startswith '/account/'",
        "backend_ref": "account_servers"
      }
    ]
```

Requests received on the frontend app_frontend on listener secure_listener, are forwarded to the billing_servers backend if their URL starts with */checkout*. If they are received on the same frontend but on listener http_listener instead, and their URL starts with */ad* then they are forwarded to a different backend (advert_servers).

The third route in the example specifies that regardless of the frontend/listener on which the request is received, if the URL starts with /account, then these requests will be forwarded to the account_servers backend.

We will discuss later the **filter** attribute and the valid expressions that can be specified.

> **Note:**
>
> The routes are evaluated in the order they are listed. The first matched route is the one that takes effect. If a request doesn't match any route, it gets dropped.
>
> You can also omit the filter attribute from the route if you want to match all requests from a certain frontend/listener. For example, the following route matches ALL requests from the frontend app_frontend and listener secure_listener, and forwards these to the billing_servers backend.

```
{

  "name": "checkout_route",

  "frontend_ref": "app_frontend",

  "listener_ref": "secure_listener",

  "backend_ref": "billing_servers"

}
```

If you want to specify a default backend for a given frontend, where requests not matching any route would be routed to, you can do so using a default_backend_ref attribute as part of the frontend/listener:

```
{

  "application": {

    "name": "app8",

    "frontends": [

      {

        "name": "admin_frontend",
```

```
        "virtual_ip": "70.122.23.26",
        "port": 443,
        "protocol": "HTTPS",
        "default_certificate_ref": "admin_cert",
        "default_backend_ref": "ad_servers"
      },
    ]
  }
}
```

Of course, if you are using listeners, the same setting is available on a given listener.

LoadBalancing Settings

So far, we have covered how to specify the backend servers but we haven't discussed how you can tune the load balancing settings such as the LB algorithm or the persistence settings. This is handled by the load_balancing attribute.

```
{
  "application": {
    "name": "app4",
    "virtual_ip": "70.122.23.11",
    "protocol": "HTTPS",
    "default_certificate_ref": "app1_cert",
    "servers": [
```

```
      "192.168.10.11",
      "192.168.10.14"
    ],
    "load_balancing": {
      "algorithm": "LEASTCONNECTION",
      "stickiness_type": "SOURCEIP",
      "stickiness_timeout": "2hr"
    },
    "unavailable_redirect_url": "https://www.example.com"
  }
}
```

The algorithm specifies the load balancing algorithm used to load balance traffic to the servers of this application. The stickiness_type specifies the persistence setting. In this case it is setup to SOURCE IP, which means that once a server is selected to serve a request from a specific client, then all subsequent requests from the same client (same source IP) will be served by the same server. The stickiness timeout specifies the duration for honoring the stickiness to the same server if a client becomes inactive for that period. The last setting is a redirect URL to send users to, if the application becomes unavailable (all servers are down).

If you have multiple backends in your application, you can define different load-balancing settings for each backend as shown in this example:

```
{
  "application": {
    "name": "app5",
    "virtual_ip": "70.122.23.11",
```

```
"protocol": "HTTPS",
"default_certificate_ref": "portal_cert",
"backends": [
  {
    "name": "account_servers",
    "port": 8080,
    "servers": [
      "192.168.10.11",
      "192.168.10.14"
    ],
    "load_balancing": {
        "algorithm": "ROUNDROBIN",
        "stickiness_type": "COOKIEINSERT"
    }
  },
  {
    "name": "billing_servers",
    "port": 8080,
    "servers": [
        "192.168.11.11",
        "192.168.11.14"
    ],
```

```
        "load_balancing": {
            "algorithm": "LEASTCONNECTION",
            "stickiness_type": "SOURCEIP"
        }
      }
    ],
    "routes": [
      {
        "name": "account_route",
        "filter": "http.request.uri startswith '/account/'",
        "backend_ref": "account_servers"
      },
      {
        "name": "checkout_route",
        "filter": "http.request.uri contains '/checkout/'",
        "backend_ref": "billing_servers"
      }
    ]
  }
}
```

If you want to share the same load_balancing settings between multiple backends, you can define them once and reference them in multiple backends. The following example shows the definitions specified in the element load_balancing_defs. From each backend, we can reference the settings to use by using the attribute load_balancing_ref.

```
{

  "application": {

    "name": "app5",

    "virtual_ip": "70.122.23.11",

    "protocol": "HTTPS",

    "default_certificate_ref": "portal_cert",

    "backends": [

      {

        "name": "account_servers",

        "port": 8080,

        "servers": [

          "192.168.10.11",

          "192.168.10.14"

        ],

        "load_balancing_ref": "standard_settings"

      },

      {

        "name": "billing_servers",

        "port": 8080,
```

```
      "servers": [
        {
          "ip": "192.168.11.11",
          "weight": 3
        },
        {
          "ip": "192.168.11.14"
        }
      ],
      "load_balancing_ref": "experimental_settings"
    }
  ],
  "routes": [
    {
      "name": "account_route",
      "filter": "http.request.uri startswith '/account/'",
      "backend_ref": "account_servers"
    },
    {
      "name": "checkout_route",
      "filter": "http.request.uri contains '/checkout/'",
      "backend_ref": "billing_servers"
```

```
      }
    ],
    "load_balancing_defs": [
      {
        "name": "standard_settings",
        "algorithm": "ROUNDROBIN",
        "stickiness_type": "SOURCEIP"
      },
      {
        "name": "experimental_settings",
        "algorithm": "LEASTCONNECTION",
        "stickiness_type": "COOKIEINSERT"
      }
    ]
  }
}
```

Note:

If you don't specify any load_balancing settings, then the default algorithm (LEASTCONNECTION) is used, and no stickiness is configured.

Certificates

You might have noticed that in many previous examples, we have certificate references in our application payloads, such as in the following simple application example:

```
{

  "application": {

    "name": "app3",

    "virtual_ip": "70.122.23.11",

    "default_certificate_ref": "app3_cert",

    "servers": [

      "192.168.10.11",

      "192.168.10.14"

    ]

  }

}
```

The certificate app3_cert is a certificate that would have already been uploaded to NetScaler using the following Next-Gen API:

```
POST /mgmt/api/nextgen/v1/certificates



{

  "certificate": {

    "name": "app1_cert",

    "public_cert": "-----BEGIN CERTIFICATE-----  … -----END
```

```
CERTIFICATE-----\n",

    "private_key": "-----BEGIN PRIVATE KEY----- … -----END PRIVATE
KEY-----\n",

    "passphrase": "long secret phrase",

    "ca_certs": [

      "-----BEGIN CERTIFICATE----- … -----END CERTIFICATE-----\n",

      "-----BEGIN CERTIFICATE----- … -----END CERTIFICATE-----\n"

    ]

  }

}
```

In this API, you specify:

- **public_cert**: Contents of the certificate public key in PEM format.
- **private_key**: Contents of the certificate private key in PEM format.
- **passphrase**: The passphrase used to decrypt the private key if one is used.
- **ca_certs**: The intermediate/root certificates chain in PEM format. Each certificate should be in a separate string in this list.

In order to transform a PEM file to a JSON string so you can use it in the API payloads, you need to replace all the line feeds with the "\\n" character.

Note:

You can use the jq utility to help easily transform a PEM file to a JSON string:

```
> jq -sR . mycert.pem
```

If the API of uploading the certificate is successful, then you can make use of it by referencing your uploaded certificate when you create your application.

```
{
  "application": {
    "name": "app3",
    "virtual_ip": "70.122.23.11",
    "certificate_refs": ["app3_default_cert"],
    "servers": [
      "192.168.10.11",
      "192.168.10.14"
    ]
  }
}
```

In the above payload, we used the attribute **certificate_refs** to reference the SNI certificates used by our application. The application is only served to clients who provide a matching SNI (Server Name Indication) to one of the certificates during the SSL handshake.

If you want NetScaler to serve any client regardless of whether they provide a matching SNI or not, you can use the attribute **default_certificate_ref** (that we used in previous examples) rather than **certificate_refs** as shown below:

```
{
  "application": {
    "name": "app3",
    "virtual_ip": "70.122.23.11",
    "default_certificate_ref": "app3_default_cert",
    "servers": [
```

```
      "192.168.10.11",

      "192.168.10.14"

    ]

  }

}
```

If you want your application to be associated with one or multiple certificates, one for each domain, but also to have a default certificate in case the client doesn't provide an SNI, or the client provided SNI doesn't match any of the configured certificates, then you can combine both the SNI certificates with the default certificate as shown below:

```
{

  "application": {

    "name": "app3",

    "virtual_ip": "70.122.23.11",

    "certificate_refs": ["app3_domain1_cert", "app3_domain2_cert",
"app3_domain3_cert"],

    "default_certificate_ref": "app3_default_cert",

    "servers": [

      "192.168.10.11",

      "192.168.10.14"

    ]

  }

}
```

> **Note:**

> You can only delete a certificate if it not used (referenced) by any application.

TLS Settings

You can also configure the security settings of the TLS connections such as the TLS protocol versions supported and the specific ciphers that can be negotiated between clients/servers and the NetScaler.

A simple way to set the TLS security level for your application, is to set it to "HIGH", "MEDIUM" or "LOW", which translate to settings on protocol versions and ciphers settings. For example, the following configuration sets it to HIGH.

```
{
  "application": {
    "name": "app3",
    "virtual_ip": "70.122.23.11",
    "default_certificate_ref": "app3_cert",
    "tls_settings": {
      "security_level": "HIGH",
      "session_reuse": true,
      "session_timeout": "48s"
    },
    "servers": [
      "192.168.10.11",
      "192.168.10.14"
    ]
```

```
  }

}
```

The setting "HIGH" of the **security_level** attribute translates to SSL Labs A+ rating and allows only TLS-1.3 protocol version to be negotiated, as well as the fixed list of TLS-1.3 ciphers:

- TLS1.3-AES256-GCM-SHA384
- TLS1.3-CHACHA20-POLY1305-SHA256

The setting "MEDIUM" allows both "TLS-1.2" and "TLS-1.3" protocol versions to be negotiated.

For TLS-1.2, the following ciphers are set by default:

- TLS1.2-AES-256-SHA256
- TLS1.2-AES-128-SHA256
- TLS1.2-AES256-GCM-SHA384
- TLS1.2-AES128-GCM-SHA256
- TLS1.2-ECDHE-RSA-AES-256-SHA384
- TLS1.2-ECDHE-RSA-AES-128-SHA256
- TLS1.2-ECDHE-RSA-AES256-GCM-SHA384
- TLS1.2-ECDHE-RSA-AES128-GCM-SHA256
- TLS1.2-ECDHE-ECDSA-AES256-SHA384
- TLS1.2-ECDHE-ECDSA-AES128-SHA256
- TLS1.2-ECDHE-ECDSA-AES256-GCM-SHA384
- TLS1.2-ECDHE-ECDSA-AES128-GCM-SHA256

For TLS1.3, the following ciphers are set by default in addition to the ciphers mentioned under HIGH security_level setting:

- TLS1.3-AES128-GCM-SHA256

The lowest setting is "LOW" and this allows "TLS-1", "TLS-1.2" and "TLS-1.3". Note that the use of TLS-1 is discouraged and is considered a weak protocol from a security perspective. So, only set to "LOW" if you have clients requiring the use of "TLS-1".

The following ciphers are set by default for "TLS-1":

- TLS1-AES-256-CBC-SHA
- TLS1-AES-128-CBC-SHA
- TLS1-ECDHE-RSA-AES256-SHA
- TLS1-ECDHE-RSA-AES128-SHA
- TLS1-ECDHE-ECDSA-AES256-SHA
- TLS1-ECDHE-ECDSA-AES128-SHA
- TLS1-DHE-RSA-AES-256-CBC-SHA
- TLS1-DHE-RSA-AES-128-CBC-SHA
- TLS1-DHE-DSS-AES-256-CBC-SHA
- TLS1-DHE-DSS-AES-128-CBC-SHA

For TLS1.2 and TLS1.3, default cipher list is inherited from HIGH and MEDIUM security_level settings.

Older SSL protocols (SSLv3 and below) are not supported in the Next-Gen API.

You can also configure **tls_settings** on a frontend. Different frontends of an application could have different TLS settings.

If you want more granular control over the protocol versions and ciphers, you can specify the settings as follows:

```
{

  "tls_settings": {

    "tls_protocols": ["TLS-1"],

    "tls_cipher_suites": [

      "TLS1-EXP1024-RC4-SHA",

      "TLS1-EXP1024-DHE-DSS-DES-CBC-SHA",

      "TLS1-EXP1024-DHE-DSS-RC4-SHA",

      "TLS1-EXP1024-RC4-MD5",
```

```
      "TLS1-EXP1024-RC2-CBC-MD5"
    ],
    "session_reuse": true,
    "session_timeout": "48s"
  }
}
```

This allows you to control exactly what protocol and ciphers you want to allow, as well as configure other TLS settings such as session reuse and session timeout:

> **Note:**
>
> If TLS Settings are not specified for an application, the default security level configured for the application is MEDIUM.

Health Checks

One of the main functions that NetScaler provides is the ability to monitor the health of your application servers. NetScaler provides rich functionality when it comes to health monitoring, with different types of health checks based on the application protocol.

Next-Gen API makes the use of this functionality simple and straightforward. You can currently specify the following health checks:

- PING (ICMP)
- HTTP

A simple health check of the application servers could consist of a regular PING to the servers to check their health. This can be configured as follows:

```
{
  "application": {
```

```
    "name": "app3",
    "virtual_ip": "70.122.23.11",
    "default_certificate_ref": "app3_cert",
    "servers": [
      "192.168.10.11",
      "192.168.10.14"
    ],
    "health_check": {
        "type": "PING"
    }
  }
}
```

The health check settings all have default values if not specified. For example, the regular interval of PINGs sent to the backend server will default to the value of 5 seconds. If you want to change it to a different value, you can specify it as follows:

```
{
  "application": {
    "name": "app3",
    "virtual_ip": "70.122.23.11",
    "default_certificate_ref": "app3_cert",
    "servers": [
      "192.168.10.11",
```

```
      "192.168.10.14"

    ],

    "health_check": {

       "type": "PING",

       "interval": "10s"

    }

  }

}
```

For the HTTP health check, any probe request that returns an HTTP status code 200 is considered a healthy response. You can for example further tailor the behavior of the health check from the default behavior by specifying other attributes:

```
{

  "application": {

    "name": "app3",

    "virtual_ip": "70.122.23.11",

    "protocol": "HTTPS",

    "default_certificate_ref": "app3_cert",

    "servers": [

      "192.168.10.11",

      "192.168.10.14"

    ],

    "health_check": {
```

```
        "interval": "5s",
        "type": "HTTP",
        "probe_request_str": "GET /health",
        "probe_healthy_response": {
             "status_codes": ["200-399"]
        },
        "num_retries_on_failure": "3",
        "num_retries_on_success": "5"
    }
  }
}
```

You can also specify multiple health checks for the same backend of servers. This means that if any of the health checks fails on the backend, the application is marked down.

```
{
  "application": {
    "name": "app3",
    "virtual_ip": "70.122.23.11",
    "default_certificate_ref": "app3_cert",
    "servers": [
      "192.168.10.11",
      "192.168.10.14"
```

```
    ],
    "health_checks": [
       {
            "name": "ping_mon",
            "type": "PING"
       },
       {
            "name": "http_mon",
            "type": "HTTP",
            "interval": "5s",
            "probe_request": {
                "path": "/health"
            }
       }
    ]
  }
}
```

You can explore all the health check settings by referring to the OpenAPI specification of Next-Gen API.

Rate Limiting

The rate limiting feature of NetScaler enables you to set maximum traffic loads for your application and take real-time preventive actions based on traffic rates.

Next-Gen API makes the use of this functionality simple and straightforward.

A simple rate limiting configuration for an application could involve setting a maximum number of requests allowed within a specific time period. This can be configured as follows:

```
{
  "application": {
    "name": "app3",
    "virtual_ip": "70.122.23.11",
    "default_certificate_ref": "app3_cert",
    "servers": [
      "192.168.10.11",
      "192.168.10.14"
    ],
    "rate_limit_settings": {
      "name": "rate_limit_1",
      "max_requests": 10,
      "time_period": "100ms",
      "action": "DROP"
    }
  }
}
```

In the following example, the rate limiting settings specify that a maximum of 10 requests are allowed within a 100 ms time period. If this limit is exceeded, the requests will be dropped.

Additionally, you can further refine the settings to apply rate limiting specifically to requests with the URL abc.com.

```
{
  "application": {
    "name": "app3",
    "virtual_ip": "70.122.23.11",
    "default_certificate_ref": "app3_cert",
    "servers": [
      "192.168.10.11",
      "192.168.10.14"
    ],
    "rate_limit_settings": {
      "name": "rate_limit_1",
      "max_requests": 10,
      "time_period": "100ms",
      "action": "DROP",
      "additional_settings": {
        "selectors": [
          {
            "field": "URL",
            "value": "abc.com"
          }
```

```
        ]
      }
    }
  }
}
```

You can also configure rate limiting to distribute the load uniformly across a 10ms time slice. In this case, if there are more than 1 request within a 10 ms window, the rate limiting action is applied. By default, the limit_type is 'BURSTY', meaning the rate limiting action is applied when the maximum number of requests is breached within the specified time period. To achieve uniform distribution, you need to set the limit_type field to 'SMOOTH'.

```
{
  "application": {
    "name": "app3",
    "virtual_ip": "70.122.23.11",
    "default_certificate_ref": "app3_cert",
    "servers": [
      "192.168.10.11",
      "192.168.10.14"
    ],
    "rate_limit_settings": {
      "name": "rate_limit_1",
      "max_requests": 10,
      "time_period": "100ms",
```

```
      "action": "DROP",
      "limit_type": "SMOOTH"
    }
  }
}
```

HTTP Responder

To configure NetScaler to respond to client requests on behalf of the backend servers for certain requests, use the HTTP Responder feature.

Case 1

Using the `raw_response` field, you can configure NetScaler to return a raw response for requests that match the expression specified as a filter in the `http_responder` field.

In the following example, NetScaler responds with a "'404' Not Found" error for any request whose URL contains the "admin" string.

```
{
  "application": {
    "name": "app3",
    "virtual_ip": "70.122.23.11",
    "default_certificate_ref": "app3_cert",
    "servers": [
      "192.168.10.11",
      "192.168.10.14"
    ],
```

```
    "http_responder": {
      "name": "admin_response",
      "filter": "http.request.uri contains 'admin'",
      "response": {
        "raw_response": "'404' Not Found"
      }
    }
  }
}
```

Case 2

Using the `responder_html_page_ref` field, you can configure NetScaler to return the content of an HTML page for requests that match the expression specified as a filter in the `http_responder` field. To do so, you must first create an HTML page by using the following request:

```
POST /mgmt/api/nextgen/v1/responder_html_page/page1



{
  "responder_html_page": {
    "name": "page1",
    "content":
"PGh0bWw+PGhlYWQ+PHRpdGxlPkV4YW1wbGU8L3RpdGxlPjwvaGVhZD48Ym9keT48aDE+SGVsb
G8sIFdvcmxkITwvaDE+PC9ib2R5PjwvaHRtbD4="
  }
```

```
}
```

In the following example, NetScaler responds with the HTML content in page1 for any request whose URL contains the string “admin”.

```
{
  "application": {
    "name": "app3",
    "virtual_ip": "70.122.23.11",
    "default_certificate_ref": "app3_cert",
    "servers": [
      "192.168.10.11",
      "192.168.10.14"
    ],
    "http_responder": {
      "name": "admin_response",
      "filter": "http.request.uri contains 'admin'",
      "response": {
        "responder_html_page_ref": "page1"
      }
    }
  }
}
```

The HTTP Responder settings can be scoped to handle requests to the whole application, to a frontend, to a listener or to a backend.

Request Transformation and Response Transformation

NetScaler provides powerful capabilities to transform both requests and responses. This feature allows you to modify the content of HTTP requests and responses to meet specific requirements, such as URL rewriting, header manipulation, and content modification.

Example: URL Rewriting

In this example, we rewrite the URL of incoming requests from `/old-path` to `/new-path`.

```
{

  "application": {

    "name": "app3",

    "virtual_ip": "70.122.23.11",

    "default_certificate_ref": "app3_cert",

    "servers": [

      "192.168.10.11",

      "192.168.10.14"

    ],

    "request_transform": {

      "name": "admin_response",

      "filter": "http.request.uri.path equals '/old-path'",

      "action": "replace_url_path",

      "url_path": "/new-path"

    }
```

```
  }
}
```

HTTP Settings

The HTTP protocol has evolved over the years to contain a lot of features that address performance and security needs. NetScaler being an L7 proxy, one of the popular features is the ability to have fine-grained control over the processing of HTTP protocol and client connections, what features of HTTP are enabled/disabled, and set various optimization and security settings.

The following payload shows an application that is explicitly configuring some HTTP settings under the http_config section (all the possible settings that can be configured are shown in this example)

```
{
  "application": {
    "name": "app10",
    "frontends": [
      {
        "name": "app_frontend",
        "virtual_ip": "71.122.24.21",
        "listeners": [
          {
            "name": "secure_listener",
            "port": 443,
            "protocol": "HTTPS",
            "default_certificate_ref": "app3_cert",
```

```
"http_to_https_redirect": true,
"http_redirect_port": 81,
"http_config": {
  "name": "app_http_settings",
  "safety_settings": {
    "drop_invalid_requests": false,
    "drop_extra_crlf_lines_after_headers": true,
    "drop_http_trace_requests": false,
    "drop_noncompliant_rfc7230_requests": false,
    "drop_connect_requests": false,
    "drop_http_header_with_extra_whitespace": false,
    "drop_extra_data_exceeding_content_length": false,
    "max_header_length": "24820B",
    "timeout_for_requests_with_incomplete_headers": "7s",
    "timeout_for_receiving_complete_requests": "10s",
    "adapt_timeout_value_based_on_traffic_conditions": false,
    "action_on_incomplete_requests": "DROP"
  },
  "optimization_settings": {
    "enable_reuse_pool_of_server_connections": true,
    "reuse_pool_settings": {
      "min_connections_in_pool": 0,
```

```
      "max_connections_in_pool": 0,
      "timeout_for_idle_connections": "1hr"
    },
    "max_requests_per_connection": 0,
    "start_cmp_on_tcp_push_flag": false
  },
  "allowed_protocols": {
    "websocket": false,
    "http2": false,
    "rtsp": false
  },
  "http2_settings": {
    "strict_cipher_selection": false,
    "max_header_list_size": "24576B",
    "max_frame_size": "16384B",
    "max_concurrent_streams": 100,
    "initial_window_size_for_connections": "65535B",
    "initial_window_size_for_streams": "65535B",
    "max_header_cmp_table_size": "4096B",
    "min_server_connections_before_multiplexing": 20,
    "max_ping_frames_per_minute": 60,
    "max_settings_frames_per_minute": 15,
```

```
            "max_reset_frames_per_minute": 90,
            "max_empty_frames_per_minute": 60
          },
          "grpc_settings": {
            "enable_grpc_length_delimitation": true,
            "max_size_for_packet_buffering": "131072B",
            "max_time_for_packet_buffering": "1s"
          },
          "misc_settings": {
            "web_logging": true,
            "generate_adc_etag_for_response_etag": false
          }
        }
      },
      {
        "name": "http_listener",
        "port": 80,
        "protocol": "HTTP"
      }
    ]
  }
],
```

```
"backends": [
  {
    "name": "advert_servers",
    "servers_port": 8080,
    "servers": [
      "192.168.10.11",
      "192.168.10.14"
    ]
  },
  {
    "name": "billing_servers",
    "servers_port": 8080,
    "servers": [
      {
        "ip": "192.168.11.11",
        "weight": 3
      },
      {
        "ip": "192.168.11.14"
      }
    ]
  }
```

```
    ],
    "routes": [
      {
        "name": "checkout_route",
        "filter": "http.request.uri contains '/checkout/'",
        "frontend_ref": "app_frontend",
        "listener_ref": "secure_listener",
        "backend_ref": "billing_servers"
      },
      {
        "name": "ad_route",
        "filter": "http.request.uri contains '/ad/'",
        "frontend_ref": "app_frontend",
        "listener_ref": "http_listener",
        "backend_ref": "advert_servers"
      }
    ]
  }
}
```

Logging

Another important aspect of configuring the NetScaler device is audit logging. To enable user-defined logging, use the following CLI command:

```
set syslogparams -userDefinedAuditlog YES
```

Logging can be integrated with various network functions such as routes, HTTP redirects, HTTP responders, request transformations, and response transformations.

In the following example, logging is incorporated as part of the HTTP responder network function:

```
{

  "application": {

    "name": "app3",

    "virtual_ip": "70.122.23.11",

    "default_certificate_ref": "app3_cert",

    "servers": [

      "192.168.10.11",

      "192.168.10.14"

    ],

    "http_responder": {

      "name": "admin_response",

      "filter": "http.request.uri contains 'admin'",

      "response": {

        "raw_response": "'404' Not Found"

      },
```

```
      "audit_log_ref": "log_port"

    },

    "audit_log_defs": [

      {

        "name": "log_port",

        "log_level": "INFORMATIONAL",

        "message": "URL={{HTTP.HOST}}{{HTTP.REQUEST.URI}} SOURCE-
PORT={{HTTP.REQUEST.TCP.SRCPORT}} DESTINATION-
PORT={{HTTP.REQUEST.TCP.DSTPORT}}"

      }

    ]

  }

}
```

Wireshark Expressions

NetScaler features a powerful policy infrastructure language called Policy Infrastructure eXpression Language (PIXL) that allows for expressing rich filters on traffic in order to select a subset of traffic. Then various optimizations and security features available on the NetScaler can be applied to that subset.

In NetScaler configuration through CLI, Nitro API or NetScaler GUI, PIXL expressions are used in NetScaler policy rules and actions. The PIXL language flexibility and richness comes at the cost of ease of use, as users need to learn the syntax and structure of this language. For example, if we want to select HTTP URLs whose path start with "/home", we would need to write the following PIXL expression:

```
HTTP.REQ.URL.HOSTNAME.EQ("www.cloud.com")
```

Or if we want to select requests for which the user-agent header contains a certain string, the PIXL expression is going to be slightly more involved:

```
HTTP.REQ.HEADER("user-agent").CONTAINS("mobile")
```

In order to provide an easier language for filtering networking traffic in Next-Gen API, we adopted the Wireshark display filters language, which is more intuitive and well familiar to many networking professionals. For example, both examples above can be expressed respectively using the following 2 Wireshark expressions :

```
http.host == "www.cloud.com"
```

or

```
http.host eq "www.cloud.com"
```

and

```
http.request.header.user-agent contains "mobile"
```

This relative simplicity is more apparent when the expression requires combining multiple conditions with "and" and "or" logical operators. The following PIXL expression:

```
HTTP.REQ.URL.HOSTNAME.EQ("www.cloud.com") && HTTP.REQ.HEADER("user-
agent").CONTAINS("mobile")
```

can be expressed as a Wireshark filter expression more simply:

```
http.host == "www.cloud.com" and http.request.header.user-agent contains
"mobile"
```

If we want to compare regardless of case sensitivity, we can use the lower() or upper() helper functions as follows:

```
lower(http.host) == "www.cloud.com" and upper(http.request.header.user-
agent) contains "MOBILE"
```

Note:

Currently, http.request expression with following attributes is supported:

- Request Hostname (http.host)
- Request Whole URL (http.request.uri)
- Request URL Path (http.request.uri.path)
- Request URL Query String (http.request.uri.query)
- Request HTTP Method (http.request.method)
- Request HTTP Cookie (http.request.header.cookie),
- Request HTTP Headers (http.request.header.\<header-name>)

In terms of operators that can be used in expressions, the following operators are supported:

- Equality: eq, ==
- Inequality: ne, !=
- Greater or Equal: >=, ge
- Lesser or Equal: <=, le
- Greater: >, gt
- Lesser: <, lt
- Substring test: contains
- Substring test: startswith
- Substring test: endswith
- Converting to uppercase: upper
- Converting to lowercase: lower
- Logical AND: and
- Logical OR: or

When we need to compare a field for a match against multiple values, one way to do this would be:

http.host == "www.site1" or http.host == "www.site2" or http.host == "www.site3"

This can lead to large and complex wireshark expressions if the list of values is large. Next-Gen API introduces an extension to wireshark expressions that allows you to do the following:

http.host **equals_any** *valid_hostnames*

Where *valid_hostnames* is a value set. Value sets are equivalent in function to NetScaler Patsets and Datasets use in Policy Expressions.

You can create value sets in Next-Gen API using the resource: /mgmt/api/nextgen/v1/filters/value_sets

For example, the following API creates the value set *valid_hostnames* that was used in the previous wireshark expression:

```
POST /mgmt/api/nextgen/v1/filters/value_sets



{

  "value_set": {

    "name": "valid_hostnames",

    "type": "string",

    "values": [

      "www.site1",

      "www.site2",

      "www.site3"

    ]

  }

}
```

The type attribute of a value set can take 4 possible values: string, number, ipv4, ipv6.

In wireshark expressions, the value sets use is possible through the following operators:

- equals_any
- contains_any
- startswith_any
- endswith_any

For users who prefer using NetScaler PIXL expressions, they can still do so by setting the attribute "filter_format" to "pixl". For example, the following configuration for an HTTP Responder uses PIXL rather a Wireshark expression:

```
{

    "http_responder": {

     "name": "admin_response",

     "filter": "HTTP.REQ.URL.PATH_AND_QUERY.CONTAINS(\"admin\")",

     "filter_format": "pixl",

     "response": "'404' Not Found"

    }

}
```

Which is equivalent to the following configuration that uses the wireshark filter expressions:

```
{

    "http_responder": {

    "name": "admin_response",

     "filter": "http.request.uri contains 'admin'",

     "response": "'404' Not Found"

    }
```

```
}
```

Note:

Currently, when you use PIXL expressions in your API, Next-Gen API cannot validate these expressions until they are installed on the NetScaler data plane. This means that you have to query the config status to ensure that your PIXL expressions are valid and that the application configuration has been successfully accepted by the data plane.

HTTP Callouts

HTTP callouts allow NetScaler to send HTTP or HTTPS requests to an external application when specific criteria are met during policy evaluation. Using HTTP callouts, you can do the following tasks: stall policy evaluation, retrieve information, update databases, or modify web server content. To configure an HTTP callout, set up an application (HTTP callout agent) on the server to respond to the callout request, ensuring consistent response formats. Then, configure the HTTP callout on NetScaler as shown below:

```
POST /mgmt/api/nextgen/v1/http_callouts/is_ip_allowed



{

  "http_callout": {

    "name": "is_ip_allowed",

    "callout_request": {

      "method": "POST",

      "scheme": "HTTPS",

      "host": "1.2.3.4",

      "port": 443,
```

```
    "path_expression": "/check_valid_ip",
    "headers": [
      {
        "name": "Version",
        "value_expression": "1.0.0"
      },
      {
        "name": "Content-Type",
        "value_expression": "application/json"
      },
      {
        "name": "api-key",
        "value_expression": "8395-3c22-abj4-0a29"
      }
    ],
    "body_expression": "{\"client_ip\": \"{{http.request.ip.src}}\"}"
  },
  "callout_response": {
    "return_value_expression": "http.response.code == 200",
    "return_value_datatype": "boolean",
    "cache_duration": "1min"
  }
```

```
  }
}
```

The HTTP callout can be used in filter expressions as shown below:

```
{
    "http_responder": {
     "name": "admin_response",
     "filter": NOT HTTP_CALLOUT(is_ip_allowed),
     "response": {
      "raw_response": "'404' Not Found"
     }
    }
}
```

Application Health

To get the health of an application, use the following API call:

```
GET /mgmt/api/nextgen/v1/applications/{application-name}/health
```

This API call retrieves the health status of the specified application. The response includes the configured and operational states of the application, its frontends, listeners, backends, and servers. The response format differs according to the application types. Application's state is determined by the frontend & backend states. If there is at least one route where the frontend and backend are UP, then the app's `operational_state` is marked as UP

Here is an example request and response for an application with multiple backends.

Example Request

```
GET /mgmt/api/nextgen/v1/applications/my_app/health
```

Example Response


```
{

  "application": {

    "name": "my_app",

    "virtual_ip": "70.122.23.11",

    "configured_state": "ENABLED",

    "operational_state": "UP",

    "backends": [

      {

        "name": "be1",

        "configured_state": "ENABLED",

        "operational_state": "UP",

        "servers": [

          {

            "ip": "10.102.201.166",

            "port": 80,

            "configured_state": "ENABLED",

            "operational_state": "UP"

          },
```

```
      {
        "ip": "10.102.201.167",
        "port": 80,
        "configured_state": "ENABLED",
        "operational_state": "UP"
      }
    ]
  },
  {
    "name": "be2",
    "configured_state": "ENABLED",
    "operational_state": "UP",
    "servers": [
      {
        "ip": "10.102.201.70",
        "port": 81,
        "configured_state": "ENABLED",
        "operational_state": "UP",
        "health_checks": [
          {
            "name": "http_monitor",
            "operational_state": "UP",
```

```
                "total_probes": 518,

                "failed_probes": 0,

                "last_response_status": "Success - HTTP response code 200
received.",

                "last_response_time": "0.683ms"

              }

            ]

          }

        ]

      }

    ]

  }

}
```


Application Statistics

To get the statistics of an application, use the following API call:

```
GET /mgmt/api/nextgen/v1/applications/{application-name}/statistics
```

The response includes the statistics of the application, its frontends, listeners, backends, and servers. The response format differs according to the application types.

Here is an example request and response for an application with multiple backends.

Example Request

```
GET /mgmt/api/nextgen/v1/applications/my_app/statistics
```

Example Response


```
{

  "application": {

    "name": "my_app",

    "virtual_ip": "70.122.23.11",

    "configured_state": "ENABLED",

    "operational_state": "UP",

    "hits": 23,

    "hits_per_sec": 0,

    "requests": 23,

    "requests_per_sec": 0,

    "responses": 23,

    "responses_per_sec": 0,

    "request_bytes": 16748,

    "request_bytes_per_sec": 0,

    "response_bytes": 295156,

    "response_bytes_per_sec": 0,

    "packets_rcvd": 199,

    "packets_rcvd_per_sec": 0,

    "packets_sent": 307,

    "packets_sent_per_sec": 0,

    "current_client_connections": 1,
```

```
"current_client_est_connections": 0,

"current_server_connections": 0,

"current_persistence_sessions": 0,

"current_backup_persistence_sessions": 0,

"spill_over_hits": 0,

"labeled_connection": 0,

"push_labeled_connection": 0,

"deferred_request": 0,

"deferred_request_per_sec": 0,

"invalid_request_or_response": 0,

"invalid_request_or_response_dropped": 0,

"down_backup_hits": 0,

"current_multipath_tcp_sessions": 0,

"current_multipath_tcp_subflows": 0,

"listeners": [

  {

    "name": "https_listener",

    "protocol": "HTTPS",

    "port": 443,

    "configured_state": "ENABLED",

    "operational_state": "UP",

    "hits": 0,
```

```
        "hits_per_sec": 0,

        "requests": 0,

        "requests_per_sec": 0,

        "responses": 0,

        "responses_per_sec": 0,

        "request_bytes": 4740,

        "request_bytes_per_sec": 0,

        "response_bytes": 6064,

        "response_bytes_per_sec": 0,

        "packets_rcvd": 22,

        "packets_rcvd_per_sec": 0,

        "packets_sent": 22,

        "packets_sent_per_sec": 0,

        "current_client_connections": 0,

        "current_client_est_connections": 0,

        "current_server_connections": 0,

        "current_persistence_sessions": 0,

        "current_backup_persistence_sessions": 0,

        "spill_over_threshold": 0,

        "spill_over_hits": 0,

        "labeled_connection": 0,

        "push_labeled_connection": 0,
```

```
        "deferred_request": 0,
        "deferred_request_per_sec": 0,
        "invalid_request_or_response": 0,
        "invalid_request_or_response_dropped": 0,
        "down_backup_hits": 0,
        "current_multipath_tcp_sessions": 0,
        "current_multipath_tcp_subflows": 0,
        "apdex_for_client_response_times": 1,
        "average_client_ttlb": 0
      },
      {
        "name": "http_listener",
        "protocol": "HTTP",
        "port": 80,
        "configured_state": "ENABLED",
        "operational_state": "UP",
        "hits": 23,
        "hits_per_sec": 0,
        "requests": 23,
        "requests_per_sec": 0,
        "responses": 23,
        "responses_per_sec": 0,
```

```
        "request_bytes": 12008,

        "request_bytes_per_sec": 0,

        "response_bytes": 289092,

        "response_bytes_per_sec": 0,

        "packets_rcvd": 177,

        "packets_rcvd_per_sec": 0,

        "packets_sent": 285,

        "packets_sent_per_sec": 0,

        "current_client_connections": 1,

        "current_client_est_connections": 0,

        "current_server_connections": 0,

        "current_persistence_sessions": 0,

        "current_backup_persistence_sessions": 0,

        "spill_over_threshold": 0,

        "spill_over_hits": 0,

        "labeled_connection": 0,

        "push_labeled_connection": 0,

        "deferred_request": 0,

        "deferred_request_per_sec": 0,

        "invalid_request_or_response": 0,

        "invalid_request_or_response_dropped": 0,

        "down_backup_hits": 0,
```

```
        "current_multipath_tcp_sessions": 0,
        "current_multipath_tcp_subflows": 0,
        "apdex_for_client_response_times": 1,
        "average_client_ttlb": 0
      }
    ],
    "backends": [
      {
        "name": "colddrinks",
        "configured_state": "ENABLED",
        "operational_state": "UP",
        "hits": 10,
        "hits_per_sec": 0,
        "requests": 10,
        "requests_per_sec": 0,
        "responses": 10,
        "responses_per_sec": 0,
        "request_bytes": 5170,
        "request_bytes_per_sec": 0,
        "response_bytes": 287415,
        "response_bytes_per_sec": 0,
        "http2_requests": 0,
```

```
        "http2_requests_per_sec": 0,

        "http2_responses": 0,

        "http2_responses_per_sec": 0,

        "packets_rcvd": 72,

        "packets_rcvd_per_sec": 0,

        "packets_sent": 226,

        "packets_sent_per_sec": 0,

        "current_client_connections": 0,

        "current_client_est_connections": 0,

        "current_server_connections": 0,

        "current_persistence_sessions": 0,

        "current_backup_persistence_sessions": 0,

        "requests_in_surge_queue": 0,

        "spill_over_threshold": 0,

        "spill_over_hits": 0,

        "labeled_connection": 0,

        "push_labeled_connection": 0,

        "deferred_request": 0,

        "deferred_request_per_sec": 0,

        "invalid_request_or_response": 0,

        "invalid_request_or_response_dropped": 0,

        "down_backup_hits": 0,
```

```
"current_multipath_tcp_sessions": 0,
"current_multipath_tcp_subflows": 0,
"apdex_for_client_response_times": 1,
"average_client_ttlb": 0,
"num_tcpconn_reasmq_75%_reached": 0,
"num_tcpconn_reasmq_flushed": 0,
"num_server_busy_error": 0,
"num_server_busy_error_per_sec": 0,
"request_retry_count": 0,
"max_retry_count_exceeded": 0,
"servers": [
  {
    "ip": "10.146.105.23",
    "port": 80,
    "configured_state": "ENABLED",
    "operational_state": "UP",
    "health_checks": [
      {
        "name": "http_monitor",
        "operational_state": "UP",
        "total_probes": 141,
        "failed_probes": 0,
```

```
                "last_response_status": "Success - HTTP response code 200
received.",

                "last_response_time": "0.008ms"

              }

            ],

            "average_server_ttfb": 0,

            "requests": 145,

            "requests_per_sec": 0,

            "responses": 145,

            "responses_per_sec": 0,

            "request_bytes": 11163,

            "request_bytes_per_sec": 17,

            "response_bytes": 40076205,

            "response_bytes_per_sec": 81786,

            "current_client_connections": 0,

            "requests_in_surge_queue": 0,

            "current_server_connections": 11,

            "current_server_est_connections": 1,

            "connection_reuse_pool": 1,

            "num_tcpconn_reasmq_75%_reached": 0,

            "num_tcpconn_reasmq_flushed": 0

          },
```

```
          {
            "ip": "10.146.105.24",
            "port": 80,
            "configured_state": "ENABLED",
            "operational_state": "UP",
            "health_checks": [
              {
                "name": "http_monitor",
                "operational_state": "UP",
                "total_probes": 141,
                "failed_probes": 0,
                "last_response_status": "Success - HTTP response code 200
received.",
                "last_response_time": "0.657ms"
              }
            ],
            "average_server_ttfb": 0,
            "requests": 145,
            "requests_per_sec": 0,
            "responses": 145,
            "responses_per_sec": 0,
            "request_bytes": 11111,
```

```
            "request_bytes_per_sec": 17,
            "response_bytes": 40362330,
            "response_bytes_per_sec": 81786,
            "current_client_connections": 0,
            "requests_in_surge_queue": 0,
            "current_server_connections": 10,
            "current_server_est_connections": 0,
            "connection_reuse_pool": 0,
            "num_tcpconn_reasmq_75%_reached": 0,
            "num_tcpconn_reasmq_flushed": 0
        }
      ]
    },
    {
      "name": "hotdrinks",
      "configured_state": "ENABLED",
      "operational_state": "UP",
      "hits": 13,
      "hits_per_sec": 0,
      "requests": 13,
      "requests_per_sec": 0,
      "responses": 13,
```

```
        "responses_per_sec": 0,

        "request_bytes": 6838,

        "request_bytes_per_sec": 0,

        "response_bytes": 1677,

        "response_bytes_per_sec": 0,

        "http2_requests": 0,

        "http2_requests_per_sec": 0,

        "http2_responses": 0,

        "http2_responses_per_sec": 0,

        "packets_rcvd": 13,

        "packets_rcvd_per_sec": 0,

        "packets_sent": 13,

        "packets_sent_per_sec": 0,

        "current_client_connections": 0,

        "current_client_est_connections": 0,

        "current_server_connections": 0,

        "current_persistence_sessions": 0,

        "current_backup_persistence_sessions": 0,

        "requests_in_surge_queue": 0,

        "spill_over_threshold": 0,

        "spill_over_hits": 0,

        "labeled_connection": 0,
```

```
"push_labeled_connection": 0,
"deferred_request": 0,
"deferred_request_per_sec": 0,
"invalid_request_or_response": 0,
"invalid_request_or_response_dropped": 0,
"down_backup_hits": 0,
"current_multipath_tcp_sessions": 0,
"current_multipath_tcp_subflows": 0,
"apdex_for_client_response_times": 1,
"average_client_ttlb": 0,
"num_tcpconn_reasmq_75%_reached": 0,
"num_tcpconn_reasmq_flushed": 0,
"num_server_busy_error": 0,
"num_server_busy_error_per_sec": 0,
"request_retry_count": 0,
"max_retry_count_exceeded": 0,
"servers": [
  {
    "ip": "10.146.105.25",
    "port": 80,
    "configured_state": "ENABLED",
    "operational_state": "UP",
```

```
            "health_checks": [
              {
                "name": "http_monitor",
                "operational_state": "UP",
                "total_probes": "141",
                "failed_probes": "0",
                "last_response_status": "Success - HTTP response code 200
received.",
                "last_response_time": "0.766ms"
              }
            ],
            "average_server_ttfb": 0,
            "requests": 153,
            "requests_per_sec": 0,
            "responses": 153,
            "responses_per_sec": 0,
            "request_bytes": 15378,
            "request_bytes_per_sec": 17,
            "response_bytes": 93020897,
            "response_bytes_per_sec": 189835,
            "current_client_connections": 0,
            "requests_in_surge_queue": 0,
```

```
            "current_server_connections": 9,

            "current_server_est_connections": 0,

            "connection_reuse_pool": 0,

            "num_tcpconn_reasmq_75%_reached": 0,

            "num_tcpconn_reasmq_flushed": 0

          }

        ]

      }

    ]

  }

}
```

Config Sets: The Next Generation of Declarative Nitro API

Config Sets bridge the gap between the Nitro API and the Next-Gen API's declarative model. They allow you to configure advanced NetScaler features — expressed as raw Nitro entities — within the same desired-state framework as the App-Centric API. This is designed for scenarios where the feature or granularity you need is not yet natively available through the App-Centric model.

A `config_set` wraps a collection of related Nitro objects (vservers, servicegroups, servers, bindings, etc.) as a single desired-state unit. The system validates and commits the configuration synchronously, then reconciles it against the installed state asynchronously — the same behavior as App-Centric resources.

Use Case

To illustrate the config_sets API, let's configure a basic load-balancing setup with a VIP and backend servers. The config_sets API enables you to define all related objects as a single configuration unit.

```
POST /mgmt/api/nextgen/v1/config_sets

{

    "config_set": {

        "name": "app1",

        "config": {

            "server": [

                {

                    "name": "192.168.10.11",

                    "ipaddress": "192.168.10.11"
```

```
        },
        {
            "name": "192.168.10.14",
            "ipaddress": "192.168.10.14"
        }
    ],
    "servicegroup": [
        {
            "servicegroupname": "app1-_default-sg",
            "servicetype": "HTTP"
        }
    ],
    "lbvserver": [
        {
            "name": "app1-_default-lbvserver",
            "ipv46": "70.122.23.11",
            "port": 443,
            "lbmethod": "ROUNDROBIN",
            "servicetype": "SSL"
        }
    ],
    "lbvserver_servicegroup_binding": [
```

```
            {
                "name": "app1-_default-lbvserver",
                "servicegroupname": "app1-_default-sg"
            }
        ],
        "servicegroup_servicegroupmember_binding": [
            {
                "servicegroupname": "app1-_default-sg",
                "servername": "192.168.10.11",
                "port": 80
            },
            {
                "servicegroupname": "app1-_default-sg",
                "servername": "192.168.10.14",
                "port": 80
            }
        ],
        "sslvserver_sslcertkey_binding": [
            {
                "vservername": "app1-_default-lbvserver",
                "certkeyname": "app1_cert"
            }
```

```
            ]

        }

    }

}
```

Let's say we want to update the configuration to add more servers or make other changes. We can simply issue another API call to add more servers as follows:

```
PUT /mgmt/api/nextgen/v1/config_sets/app1

{

    "config_set": {

        "name": "app1",

        "config": {

            "server": [

                {

                    "name": "192.168.10.11",

                    "ipaddress": "192.168.10.11"

                },

                {

                    "name": "192.168.10.14",

                    "ipaddress": "192.168.10.14"

                },

                {

                    "name": "192.168.10.15",
```

```
                "ipaddress": "192.168.10.15"
            },
            {
                "name": "192.168.10.16",
                "ipaddress": "192.168.10.16"
            }
        ],
        "servicegroup": [
            {
                "servicegroupname": "app1-_default-sg",
                "servicetype": "HTTP"
            }
        ],
        "lbvserver": [
            {
                "name": "app1-_default-lbvserver",
                "ipv46": "70.122.23.11",
                "port": 443,
                "lbmethod": "ROUNDROBIN",
                "servicetype": "SSL"
            }
        ],
```

```
        "lbvserver_servicegroup_binding": [
            {
                "name": "app1-_default-lbvserver",
                "servicegroupname": "app1-_default-sg"
            }
        ],
        "servicegroup_servicegroupmember_binding": [
            {
                "servicegroupname": "app1-_default-sg",
                "servername": "192.168.10.11",
                "port": 80
            },
            {
                "servicegroupname": "app1-_default-sg",
                "servername": "192.168.10.14",
                "port": 80
            },
            {
                "servicegroupname": "app1-_default-sg",
                "servername": "192.168.10.15",
                "port": 80
            },
```

```
                {
                    "servicegroupname": "app1-_default-sg",
                    "servername": "192.168.10.16",
                    "port": 80
                }
            ],
            "sslvserver_sslcertkey_binding": [
                {
                    "vservername": "app1-_default-lbvserver",
                    "certkeyname": "app1_cert"
                }
            ]
        }
    }
}
```

This shows how to add two new servers (192.168.10.15 and 192.168.10.16) to the existing configuration by updating the config_set with a PUT request.

Listing Config Sets

To retrieve all config sets:

```
GET /mgmt/api/nextgen/v1/config_sets
```

Pagination

When managing a large number of config sets, use `pageno` and `pagesize` to paginate results. `pageno` specifies the page number (starting from 1) and `pagesize` specifies the number of config sets returned per page.

Retrieve the first 10 config sets:

```
GET /mgmt/api/nextgen/v1/config_sets?pageno=1&pagesize=10
```

Retrieve the second page of 10:

```
GET /mgmt/api/nextgen/v1/config_sets?pageno=2&pagesize=10
```

Count

To retrieve only the total number of config sets without fetching the full list, use `count=yes`:

```
GET /mgmt/api/nextgen/v1/config_sets?count=yes
```

Response:

```
{

    "__count": 20

}
```

Shareable Config Sets

By default, the Nitro entities defined in a config_set are exclusively owned by that config_set. Setting `shareable: true` marks those entities as shared resources that can be referenced by other config_sets without causing conflicts.

This is particularly useful for resources that are common across multiple applications — such as an SSL certificate chain. Multiple application config_sets may bind to the same certificate, but only one config_set should define and manage the underlying certificate files and certkey objects.

Marking that config_set as `shareable` allows other config_sets to coexist without attempting to take exclusive ownership of the same entities.

Example: Shared SSL certificate chain

First, create a shareable config_set that uploads the certificate files and creates the certkey objects:

```
POST /mgmt/api/nextgen/v1/config_sets

{

    "config_set": {

        "name": "cert_chain",

        "shareable": true,

        "config": {

            "systemfile": [

                {

                    "filename": "rootcert.cert",

                    "filecontent": "<base64-encoded-content>",

                    "filelocation": "/nsconfig/ssl",

                    "fileencoding": "BASE64"

                },

                {

                    "filename": "intermediate.cert",

                    "filecontent": "<base64-encoded-content>",

                    "filelocation": "/nsconfig/ssl",

                    "fileencoding": "BASE64"
```

```
        },
        {
            "filename": "servercert.cert",
            "filecontent": "<base64-encoded-content>",
            "filelocation": "/nsconfig/ssl",
            "fileencoding": "BASE64"
        },
        {
            "filename": "servercert.key",
            "filecontent": "<base64-encoded-content>",
            "filelocation": "/nsconfig/ssl",
            "fileencoding": "BASE64"
        }
    ],
    "sslcertkey": [
        {
            "certkey": "rootcert.certck",
            "cert": "rootcert.cert"
        },
        {
            "certkey": "intermediate.certck",
            "cert": "intermediate.cert",
```

```
                    "linkcertkeyname": "rootcert.certck"

                },

                {

                    "certkey": "appcert",

                    "cert": "servercert.cert",

                    "key": "servercert.key",

                    "linkcertkeyname": "intermediate.certck"

                }

            ]

        }

    }

}
```

Multiple application config_sets can now reference `appcert` in their SSL bindings without conflicting with each other:

```
POST /mgmt/api/nextgen/v1/config_sets

{

    "config_set": {

        "name": "app1",

        "config": {

            "sslvserver_sslcertkey_binding": [

                {

                    "vservername": "app1-_default-csvserver",
```

```
                    "certkeyname": "appcert"
                }
            ]
        }
    }
}
```

Config Set Statistics

The statistics endpoint returns real-time runtime statistics for the Nitro entities within a config_set — such as connection counts, request/response throughput, and health state for lbvservers and service group members.

```
GET /mgmt/api/nextgen/v1/config_sets/{config_set-name}/statistics
```

Example:

```
GET /mgmt/api/nextgen/v1/config_sets/app1/statistics
```

Response:

```
{
    "config_set": {
        "name": "app1",
        "statistics": {
            "lbvserver": [
                {
                    "name": "app1-_default-lbvserver",
```

```
                "primaryipaddress": "70.122.23.11",
                "primaryport": 443,
                "type": "SSL",
                "state": "DOWN",
                "actsvcs": "0",
                "inactsvcs": "2",
                "vslbhealth": "0",
                "tothits": "0",
                "hitsrate": 0,
                "totalrequests": "0",
                "requestsrate": 0,
                "totalresponses": "0",
                "responsesrate": 0,
                "curclntconnections": "0",
                "cursrvrconnections": "0",
                "totspillovers": "0",
                "cltresponsetimeapdex": 1.0
            }
        ],
        "servicegroup": [
            {
                "servicegroupname": "app1-_default-sg",
```

```
                    "state": "ENABLED",
                    "servicetype": "HTTP",
                    "servicegroupmember": [
                        {
                            "servicegroupname": "app1-_default-sg?
192.168.10.11?80",
                            "primaryipaddress": "192.168.10.11",
                            "primaryport": 80,
                            "state": "DOWN",
                            "totalrequests": "0",
                            "requestsrate": 0,
                            "cursrvrconnections": "0",
                            "curclntconnections": "0"
                        },
                        {
                            "servicegroupname": "app1-_default-sg?
192.168.10.14?80",
                            "primaryipaddress": "192.168.10.14",
                            "primaryport": 80,
                            "state": "DOWN",
                            "totalrequests": "0",
                            "requestsrate": 0,
                            "cursrvrconnections": "0",
```

```
                        "curclntconnections": "0"

                    }

                ]

            }

        ]

    }

}
```

Limitations

- **Statistics are only available for entity types that support them.** Entities such as `server`, `sslcertkey`, and `systemfile` do not have runtime statistics and will not appear in the response.
- **Read-only.** The statistics endpoint is GET-only — it cannot be used to reset or modify counters.

Config Set Status

When you create or update a config_set, the system validates and commits the configuration synchronously but applies it to the NetScaler data plane asynchronously. The config_status endpoint lets you track the progress of that asynchronous reconciliation.

```
GET /mgmt/api/nextgen/v1/config_sets/{config_set-name}/config_status
```

The response includes an overall `state` for the config_set and a `tasks` array describing the stages executed.

State values

STATE	DESCRIPTION
`ACCEPTED`	The config_set was received, validated, and committed to the database. Reconciliation with the data plane has not yet started or is in progress.
`INSTALLED`	The config_set has been fully applied to the NetScaler data plane. This is the expected steady state after a successful create or update.
`INSTALL_FAILED`	Applying the config_set to the data plane failed. Check the `tasks` array for which stage failed and why.
`UNINSTALL_PENDING`	A delete request has been received and is queued for processing.
`UNINSTALLED`	The config_set has been successfully removed from the data plane.
`UNINSTALL_FAILED`	Removing the config_set from the data plane failed.

Task stages

The `tasks` array captures the pipeline of operations executed during reconciliation. Each task has a `stage` and a `state`.

STAGE	DESCRIPTION
`Compute Differential Config`	The system calculates the difference between the desired state (what you submitted) and the current state on the data plane. Only the delta is applied, not the full config.
`Apply Differential Config`	The computed delta is applied to the NetScaler data plane.

STAGE	DESCRIPTION
`Compute Rollback Config`	If `Apply Differential Config` fails, the system computes a rollback config to restore the previous known-good state.
`Apply Rollback Config`	The rollback config is applied to the data plane to recover from the failed apply.

Task state values

STATE	DESCRIPTION
`PENDING`	The task has been queued but execution has not started yet.
`INPROGRESS`	The task is currently executing.
`COMPLETED`	The task finished successfully.
`DIFF_FAILED`	The diff computation failed. Only applies to the `Compute Differential Config` stage. This usually indicates a conflict or inconsistency between the desired state and the data plane state.
`ERROR`	The task encountered an error during execution. Applies to `Apply Differential Config`, `Compute Rollback Config`, and `Apply Rollback Config` stages.

Examples

Successfully installed

The config_set has been fully reconciled with the data plane:

```
{

    "status": {

        "description": "Progress status for CONFIG_SET: app1",
```

```
        "state": "INSTALLED",
        "tasks": [
            {
                "stage": "Compute Differential Config",
                "state": "COMPLETED"
            },
            {
                "stage": "Apply Differential Config",
                "state": "COMPLETED"
            }
        ]
    }
}
```

Reconciliation in progress

The config_set has been accepted and the system is computing the diff:

```
{
    "status": {
        "description": "Progress status for CONFIG_SET: app1",
        "state": "ACCEPTED",
        "tasks": [
            {
```

```
                "stage": "Compute Differential Config",
                "state": "INPROGRESS"
            }
        ]
    }
}
```

Install failed — diff computation failed

The system could not compute a valid differential config:

```
{
    "status": {
        "description": "Progress status for CONFIG_SET: app1",
        "state": "INSTALL_FAILED",
        "tasks": [
            {
                "stage": "Compute Differential Config",
                "state": "DIFF_FAILED"
            }
        ]
    }
}
```

Install failed — apply failed with rollback

The config was applied but failed on the data plane. The system attempted a rollback to restore the previous state:


```
{
    "status": {
        "description": "Progress status for CONFIG_SET: app1",
        "state": "INSTALL_FAILED",
        "tasks": [
            {
                "stage": "Compute Differential Config",
                "state": "COMPLETED"
            },
            {
                "stage": "Apply Differential Config",
                "state": "ERROR"
            },
            {
                "stage": "Compute Rollback Config",
                "state": "COMPLETED"
            },
            {
                "stage": "Apply Rollback Config",
                "state": "COMPLETED"
```

```
            }

        ]

    }

}
```

Limitations

- **Application-specific scope only.** Config Sets are intended for objects tied to a specific application. Global configuration entities are outside the scope of Config Sets and must be configured using the Nitro API directly. For example, global HTTP parameter settings cannot be configured through the Next-Gen API — see ns httpParam for the corresponding Nitro configuration.