anuta networks

atom

API Guide

version 11.9

Table Of Contents

About This Guide

ATOM is a Restconf and YANG server from a Northbound Integration point of view.

Configuring/Customizing ATOM is largely achieved by using Restconf APIs against ATOM platform Yang models, Service Models and device models.

Knowledge of YANG and Restconf are required to work with ATOM APIs.

ATOM implements (largely) the following versions

- RESTCONF: https://tools.ietf.org/html/rfc8040
- YANG 1.1: https://tools.ietf.org/html/rfc7950

There are two ways customers can integrate with ATOM.

1. Using YANG and ATOM APIs
2. Using Python SDK for Services And Device models development against ATOM APIs.

This document presents a high level view of ATOM APIs. Each Subsystem of ATOM, such as, Services Development, Device Driver Development, Workflows, Monitoring/Telemetry etc come with their individual detailed documentation.

Intended Audience

This document is meant to be used by the system integrators, IT administrators, and developers who want to develop client applications using ATOM.

Support

If you are facing any issues while using the ATOM APIs, please send a mail at support@anutanetworks.com or raise a ticket on the Support site.

Understanding ATOM APIs

Introduction

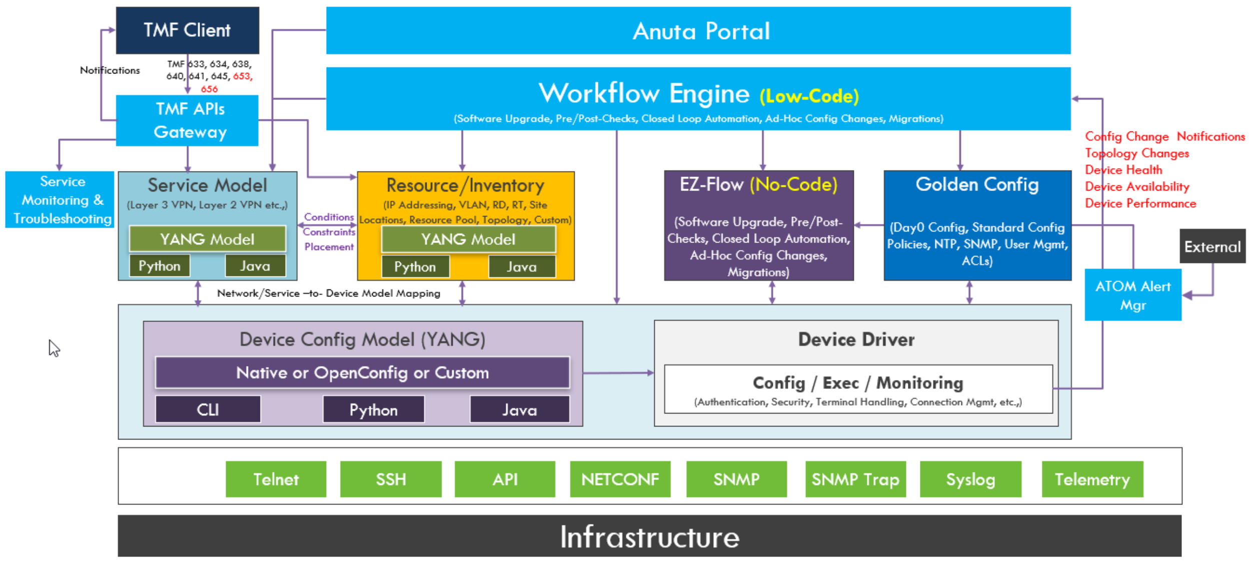
ATOM enables customers and partners to develop their own device, service and operational models for specific network service delivery needs. ATOM API implements a RESTful HTTP workflow, using HTTPS Requests to the server and retrieving the information they need from the server's Responses. Any client application that can send HTTP Requests over a secure channel using SSL through the port number 443 is an appropriate tool for developing RESTful applications with the ATOM API. The RESTCONF protocol operates on a conceptual data store defined with the YANG data modeling language. The server lists each YANG module it supports using a structure based on the YANG module capability URI format.

The conceptual data store contents, data-model-specific operations, and notification events are identified by this set of YANG module resources. All RESTCONF content identified as either a data resource, operation resource, or event stream resource is defined in YANG.

ATOM Architecture

ATOM's model-driven, layered, and abstraction approach helps in delivering vendor-neutral, extensible, and maintainable services.

For developing new services or device plugins customers can use ATOM Python SDK. Below snippet shows API interactions happening at different levels.


TMF Client
Notifications
TMF 633, 634, 638, 640, 641, 645, 653, 656
TMF APIs Gateway
Anuta Portal
Workflow Engine (Low-Code)
(Software Upgrade, Pre/Post-Checks, Closed Loop Automation, Ad-Hoc Config Changes, Migrations)
Service Monitoring & Troubleshooting
Service Model
(Layer 3 VPN, Layer 2 VPN etc.,)
YANG Model
Python
Java
Conditions
Constraints
Placement
Resource/Inventory
(IP Addressing, VLAN, RD, RT, Site Locations, Resource Pool, Topology, Custom)
YANG Model
Python
Java
EZ-Flow (No-Code)
(Software Upgrade, Pre/Post-Checks, Closed Loop Automation, Ad-Hoc Config Changes, Migrations)
Golden Config
(Day0 Config, Standard Config Policies, NTP, SNMP, User Mgmt, ACLs)
Config Change Notifications
Topology Changes
Device Health
Device Availability
Device Performance
External
ATOM Alert Mgr
Network/Service –to- Device Model Mapping
Device Config Model (YANG)
Native or OpenConfig or Custom
CLI
Python
Java
Device Driver
Config / Exec / Monitoring
(Authentication, Security, Terminal Handling, Connection Mgmt, etc.,)
Telnet
SSH
API
NETCONF
SNMP
SNMP Trap
Syslog
Telemetry
Infrastructure

ATOM Data Model

ATOM platform configuration is largely captured as YANG models. Tasks, Scheduling, Users, tenants, Services, Device models, Monitoring Infra, Actions, RBAC rules, Device Compliance Rules, Events, Notifications and so on are some of the concepts modeled with YANG 1.1.

With this arrangement coupled with Restconf as the public API it is relatively easy for customers to understand how to integrate with ATOM. The yang models can be downloaded from the server. ATOM UI also gives a schema browser and has Swagger integration to help navigate these models.

- **Device and Service Models**

  ATOM contains an abstract device model and numerous vendor devices are already mapped to this abstract model. More vendor device support can be added by supplying the relevant device package. A device package can be developed using ATOM Python SDK.

  If the device supports YANG models (native or Openconfig) ATOM can use those models instead of its abstract model.

  It is also possible to bypass the YANG model layer and use the device native cli or REST API directly.

  In addition to the device models, popular services such as L2VPN, L3VPN, Routing, ACLs etc. are available out-of-the-box.

- **Customized Operations**

  Using ATOM Data Model, the administrator can define highly customized operations that will be executed on a multi-vendor install base. Any network configuration task can be defined as an operation. Operations can be executed via API or UI with ease.

- **Transaction support with commit/rollback**

  ATOM supports transactional operations both on the model itself as well as the devices.

  ATOM supports a candidate configuration concept using which customers can stage their datastore(device configuration) changes until ready to commit. Upon commit, the staged changes will be merged to the running configuration datastore and devices are configured.

API Entities

The API model includes these programmatic entities:

- API Objects
- API Methods

API Objects

API objects represent the ATOM entities that are modeled in YANG. Programmatic access is determined by the user permissions and access settings (which are configured by your organization's system administrator). Most of the objects accessible through the API are read-write objects. However, there are a few objects that are read-only. To access the API objects, download the latest SDK containing all the objects and models expressed in YANG.

Schema Path

The schema path leads you to identify a specific target in the Data Model tree and contains the following attributes:

- A parent object
- Relative name that uniquely identifies the object among its siblings
- Schema path name that uniquely identifies the object globally

Example

```
/controller:credentials/credential-sets/credential-set
```

Instance path

The relative path identifies an object from its siblings within the context of its parent object.

Example

```
/controller:credentials/credential-sets/credential-set=XYZ
where XYZ is the name of a Credential Set
```

Object Representations

Clients must be able to send Requests and receive Responses from the objects by using either the XML or JSON data serialization formats.

The format for both the Request and Response can be specified by adding an .xml or .json extension to the Request URI or Content-Type header and the Accept header.

Format	Content-Type header	Accept header	Extension
JSON	application/json	application/json, application/yang-data+json	.json
XML	application/xml	application/xml, application/yang-data+xml	.xml

API Methods

ATOM supports the standard RESTCONF methods - PUT, PATCH, GET, POST, and DELETE. The base URL RESTCONF based operations is https://<server-ip>/restconf

HTTP Method	Operation Type	Description
GET	Retrieve	Retrieves or lists the representation of an existing object
POST	Create	Create an object
PUT	Update	Modifies the existing object
PATCH	Update	Applies the delta instead of replacing the entire object **NOTE**: ATOM supports only the plain “PATCH”method
DELETE	Delete	Deletes an existing object

URL Encodings

The target resource specified in a RESTCONF operation should adhere to the rules specified in the specification. For knowing the rules needed to encode a key value, refer RESTCONF documentation: https://tools.ietf.org/html/rfc8040#section-3.5.3

Response codes

HTTP Status Codes returned to the client in response to the request sent to the ATOM server.

Status Code	Status Description
200 OK	Successful
201 Created	The request is valid. The requested object was created.
202 Accepted	The request is valid and a task was created to handle it. Response contains the task object which can be used to monitor the task progress.
204 No Content	The request is valid and was completed. The response does not include a body.
400 Bad Request	The request body is malformed, incomplete, or otherwise Invalid.
401 Unauthorized	An authorization header was expected but not found.
403 Forbidden	The requesting user does not have adequate privileges to access one or more objects specified in the request

404 Not Found	One or more objects specified in the request could not be found in the specified container
405 Method Not Allowed	The HTTP method specified in the request is not supported for this object.
406 Not Acceptable	The requested resource is only capable of generating content which is not acceptable according to the Accept Headers sent in the request.
409 Conflict	The requested object or resource is in conflict.
415 Unsupported Media Type	The request entity has a media type which the server or resource does not support.
500 Internal Server Error	The request was received but could not be completed because of an internal error at the server

Authentication

ATOM supports bearer token based authentication. You would need a client_id, client secret, username and password to obtain a token. Your administrator can provide the client_secret by logging to ATOM access manager at <ATOM_URL>/auth with admin credentials. Navigate to Clients -> atom -> Credentials and copy the Secret. If a new client for API access is preferred, follow the steps in New Client

Token is obtained by calling the REST API by setting client_secret obtained above, and your username and password and extract the id_token from the output.

Example:

```
curl -L -k  -X POST
'<ATOM_URL>/auth/realms/system/protocol/openid-connect/token' -H
'Content-Type: application/x-www-form-urlencoded' \
--data-urlencode 'grant_type=password' \
--data-urlencode 'scope=openid' \
--data-urlencode 'client_id=<CLIENT_ID>' \
--data-urlencode 'client_secret=<CLIENT_SECRET>' \
--data-urlencode 'username=<USERNAME>' \
--data-urlencode 'password=<PASSWORD'

Output:
{"access_token":"eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJFY3RabXFUM
jZwYlU5dDdHcHh3ZWp6YV9KaENhV1V3S3hJRS1JSzVKX0RnIn0.eyJleHAiOjE2MTQzMDkyNjQsIml
hdCI6MTYxNDMwNTY2NCwianRpIjoiMTA0MGI0N2UtMzMwNi00ODc0LTlkNGEtNGU0NTNmYTQ3N2YwI
iwiaXNzIjoiaHR0cHM6Ly9hcHAuMTcyLjE2LjIyLjIwNy5uaXAuaW86MzI0NDMvYXV0aC9yZWFsbXM
vc3lzdGVtIiwiYXVkIjoiYWNjb3VudCIsInN1YiI6Ijk4NmU5MDMyLTI0Y2YtNDdhOS1hMWY5LTFjN
zJiY2RjOGE5MiIsInR5cCI6IkJlYXJlciIsImF6cCI6ImFub3RoZXItY2xpZW50Iiwic2Vzc2lvbl9
zdGF0ZSI6Ijk3MDlhZjVhLWI0ZTYtNDYwNy1hNjZjLWRmMWJmZDE4NzcxYyIsImFjciI6IjEiLCJyZ
WFsbV9hY2Nlc3MiOnsicm9sZXMiOlsib2ZmbGluZV9hY2Nlc3MiLCJ1bWFfYXV0aG9yaXphdGlvbiJ
dfSwicmVzb3VyY2VfYWNjZXNzIjp7ImFjY291bnQiOnsicm9sZXMiOlsibWFuYWdlLWFjY291bnQiL
CJtYW5hZ2UtYWNjb3VudC1saW5rcyIsInZpZXctcHJvZmlsZSJdfX0sInNjb3BlIjoib3BlbmlkIHB
yb2ZpbGUgZW1haWwiLCJlbWFpbF92ZXJpZmllZCI6ZmFsc2UsInByZWZlcnJlZF91c2VybmFtZSI6I
mFkbWluIiwiZ2l2ZW5fbmFtZSI6IiIsImZhbWlseV9uYW1lIjoiIiwiZW1haWwiOiJhZG1pbkBhdG9
tLmxvY2FsIn0.Y6Y7HaF5-Wqr7MlK_Dh9MqZuWTMPjTJ7JNOGGc7teIJm0e2I3JXokXURbZlCpolEY
jdGV2Pw12l0Lj-G1l03_A--PhdzPRinrEf0wbX3JHk8DpIoe9vKykxTMi-PmEDkYzCIuGLTBx7GTom
buft8sJ2VJejs-uoTshWd6lAW289efRQXmn_bFQhAuM_dDKHxGBOZReapf6FhMo3qDSg2Rg4mvn7_R
7X7rYMy0ko0tFn4hNCsEx-U1cbA1roH-b3GAKe_IsNtW6aneaMqegrYpNYluYvMkuY_MDXz8eX9GHS
YKg50mdeVbgh3_EFHhXQ3hDQMRej3ataWIC0zC4u6AQ","expires_in":3600,"refresh_expire
s_in":1800,"refresh_token":"eyJhbGciOiJIUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6IC
JhMzhkNTc1YS0zYTg3LTQ5Y2YtYTcwOS0xOGFlYjcxZGViYmIifQ.eyJleHAiOjE2MTQzMDc0NjQsI
mlhdCI6MTYxNDMwNTY2NCwianRpIjoiMDVjYTVmMjItNThkZi00NzUwLWE2MGMtZDBmNDdmZDAyMmN
iIiwiaXNzIjoiaHR0cHM6Ly9hcHAuMTcyLjE2LjIyLjIwNy5uaXAuaW86MzI0NDMvYXV0aC9yZWFsb
XMvc3lzdGVtIiwiYXVkIjoiaHR0cHM6Ly9hcHAuMTcyLjE2LjIyLjIwNy5uaXAuaW86MzI0NDMvYXV
0aC9yZWFsbXMvc3lzdGVtIiwic3ViIjoiOTg2ZTkwMzItMjRjZi00N2E5LWExZjktMWM3MmJjZGM4Y
TkyIiwidHlwIjoiUmVmcmVzaCIsImF6cCI6ImFub3RoZXItY2xpZW50Iiwic2Vzc2lvbl9zdGF0ZSI
6Ijk3MDlhZjVhLWI0ZTYtNDYwNy1hNjZjLWRmMWJmZDE4NzcxYyIsInNjb3BlIjoib3BlbmlkIHByb
2ZpbGUgZW1haWwifQ.XZ4QnaKY_SRKxQdmQTN72PLiTX-g_Ppwmk7FCi9Aegk","token_type":"B
earer","id_token":"eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJFY3Rab
XFUMjZwYlU5dDdHcHh3ZWp6YV9KaENhV1V3S3hJRS1JSzVKX0RnIn0.eyJleHAiOjE2MTQzMDkyN
jQsImlhdCI6MTYxNDMwNTY2NCwiYXV0aF90aW1lIjowLCJqdGkiOiI0YjUyODgxMS01OTMzLTRiY
2YtOGU3Yy03MWViYjc1MDQyYTEiLCJpc3MiOiJodHRwczovL2FwcC4xNzIuMTYuMjIuMjA3Lm5pc
C5pbzozMjQ0My9hdXRoL3JlYWxtcy9zeXN0ZW0iLCJhdWQiOiJhbm90aGVyLWNsaWVudCIsInN1Y
iI6Ijk4NmU5MDMyLTI0Y2YtNDdhOS1hMWY5LTFjNzJiY2RjOGE5MiIsInR5cCI6IklEIiwiYXpwI
joiYW5vdGhlci1jbGllbnQiLCJzZXNzaW9uX3N0YXRlIjoiOTcwOWFmNWEtYjRlNi00NjA3LWE2N
mMtZGYxYmZkMTg3NzFjIiwiYXRfaGFzaCI6IlJsV2Z1VVlCWWhsWXppc05IdTc3SmciLCJhY3IiO
iIxIiwiZW1haWxfdmVyaWZpZWQiOmZhbHNlLCJwcmVmZXJyZWRfdXNlcm5hbWUiOiJhZG1pbiIsI
mdpdmVuX25hbWUiOiIiLCJmYW1pbHlfbmFtZSI6IiIsImVtYWlsIjoiYWRtaW5AYXRvbS5sb2Nhb
CJ9.GZ9eYBaCtGesAGGvhZPBiEQ85Vtf6Ae3dP1tEVUUbZBKK_Su_B49TYO1Zw9_96hR91YHp1_4
xn-n1MEDrd54gHCtAmdZi4Woj9X9QUAWFLchQ0n2qVI0DdE4MXUTfnXopmVkGBxQSDSXoom8Fj9Q
iEBkmKM25xIFMDMGoXz75e3VXu8hHUDfCPFP2Tj5Y-oEC_s8pGbv7P7e5GLMpUG15F82_JE_VOUc
GN2BbTDolwchHD53mNRaJ2MwTIpsfQGtZkghCyulCXXgh-k7AE2jkS-BQVXB392DwgKNyDxONxdA
axNOB-wqBtNwwfXwU9sQGCh6sgo6dPr8rwroPoxvVw","not-before-policy":0,"session_st
ate":"9709af5a-b4e6-4607-a66c-df1bfd18771c","scope":"openid profile email"}
```

Extract the id_token from the above output.

Example:

```
$ export ID_TOKEN=<ID_TOKEN>

$ curl -L -k  -X GET '<ATOM_URL>/restconf/data/tenants.json' -H
"Authorization: Bearer $ID_TOKEN"

{"controller:tenants":{"@":{"shared-with":"system.*"},"tenant":[{"@":{"shared-
with":"system.*"},"sub-tenancy-enabled":false,"name":"system","description":"s
ystem
tenant","dry-run":false},{"@":{"shared-with":"system.*"},"sub-tenancy-enabled"
:false,"name":"acme","dry-run":false},{"@":{"shared-with":"system.*"},"sub-ten
ancy-enabled":false,"name":"Coke","dry-run":false},{"@":{"shared-with":"system
.*"},"sub-tenancy-enabled":false,"name":"Pepsi","dry-run":false}]}}
```

New Client

If a new client for API access is desired, follow these steps.

1. Login to the ATOM access manager using admin credentials at https://<ATOM_UI_VIP>/auth/ (e.g:https://10.10.7.30/auth)
2. Navigate to Clients, export the client atom. It would download a file
3. Click on 'Create' on the same page (Clients) and import the previous downloaded file
4. Change the client name to appropriate name
5. Click Save, it should show the new Client page.
6. (Optional) Update 'Valid Redirect URLs' to the proper callback url.
7. Select 'Credentials' and note down the Secret, this is going to be client secret.
8. Edit the oauth2-proxy deployment file by ssh to master node
    a. Execute ***kubectl edit deployment -n atom oauth2-proxy***
    b. Add https://<ATOM_UI_VIP>/auth=<client_name> (e.g: https://10.10.7.30/auth=newclient) under OAUTH2_PROXY_EXTRA_JWT_ISSUERS. If there are entries already, add the new one after a comma without a space. (e.g: https://10.10.7.30/auth=atom,https://10.10.7.30/auth=paragon-automation)

Multi Tenancy in ATOM

ATOM supports multi-tenancy by attaching the tenant owning an object as part of the overall object key. There are 2 attributes attached to all objects:

1. Owner
2. shared-with

‘system’ is always the root tenant.

Sub Tenancy is also supported.

The immediate child tenants of ‘system’ are called top level tenants.

Data of one top level tenant is completely isolated from another top level tenant.

Data of ‘system’ can be shared with top level tenants.

If tenants want to share data with ‘system’, it is possible too.

A sample hierarchy can be:

system

  Acme

    North

South

Campus-1

Company-2

Company-3

User guide has more details on Multi Tenancy Support in ATOM.

For the API interaction purposes, there is little change to the APIs to handle Multi Tenancy.

All APIs work the same when MT is enabled or not, for the most part. But, when sub tenants are involved and sharing of resources is required, sometimes, a client has to clarify which ‘owner’ of an object is being referenced. In those cases, the client program has to specify the ‘owner’ of objects as part of the payload to the APIs.

Multi Tenancy Specifics can be explained with the help of following examples:

When Multi Tenancy is disabled

This is the case when a client deploys ATOM on-premise.

In this deployment client assumes ‘system’ as self.

All the data is owned by ‘system’ [which is ‘client’ itself].

In this case there is no difference between MT and non-MT flavors of the APIs.

The rest of this document mainly describes Non-MT APIs.

Single Tenant under ‘system’

System

Acme

This scenario can happen in two cases.

In ATOM saas, when a deployment is dedicated to one client (Acme in the above example).

Or, when Acme deploys on-prem, enables Multi Tenancy and creates a top level tenant ‘Acme’.

In this scenario, there can be data sharing between system and Acme, mostly from system to Acme, to a lesser extent from Acme to system.

ATOM Deployment with Multiple Tenants but no Sub tenancy

System

Tenant-1

Tenant-2

This is the predominant style ATOM is deployed in SAAS.

But, a client can use this style in their on-prem too if they wish. Let us call this Customer SAAS, for lack of a better term.

We need to understand an important difference between ATOM SAAS and Customer SAAS.

'Anuta' is synonymous with 'system' in ATOM SAAS and

Customer is synonymous with 'system' in Customer SAAS.

Anuta employees are system users in SAAS and would not have access to tenants data (barring a few data categories). But, customers may be ok to let designated customer users see all the tenants data in their deployment.

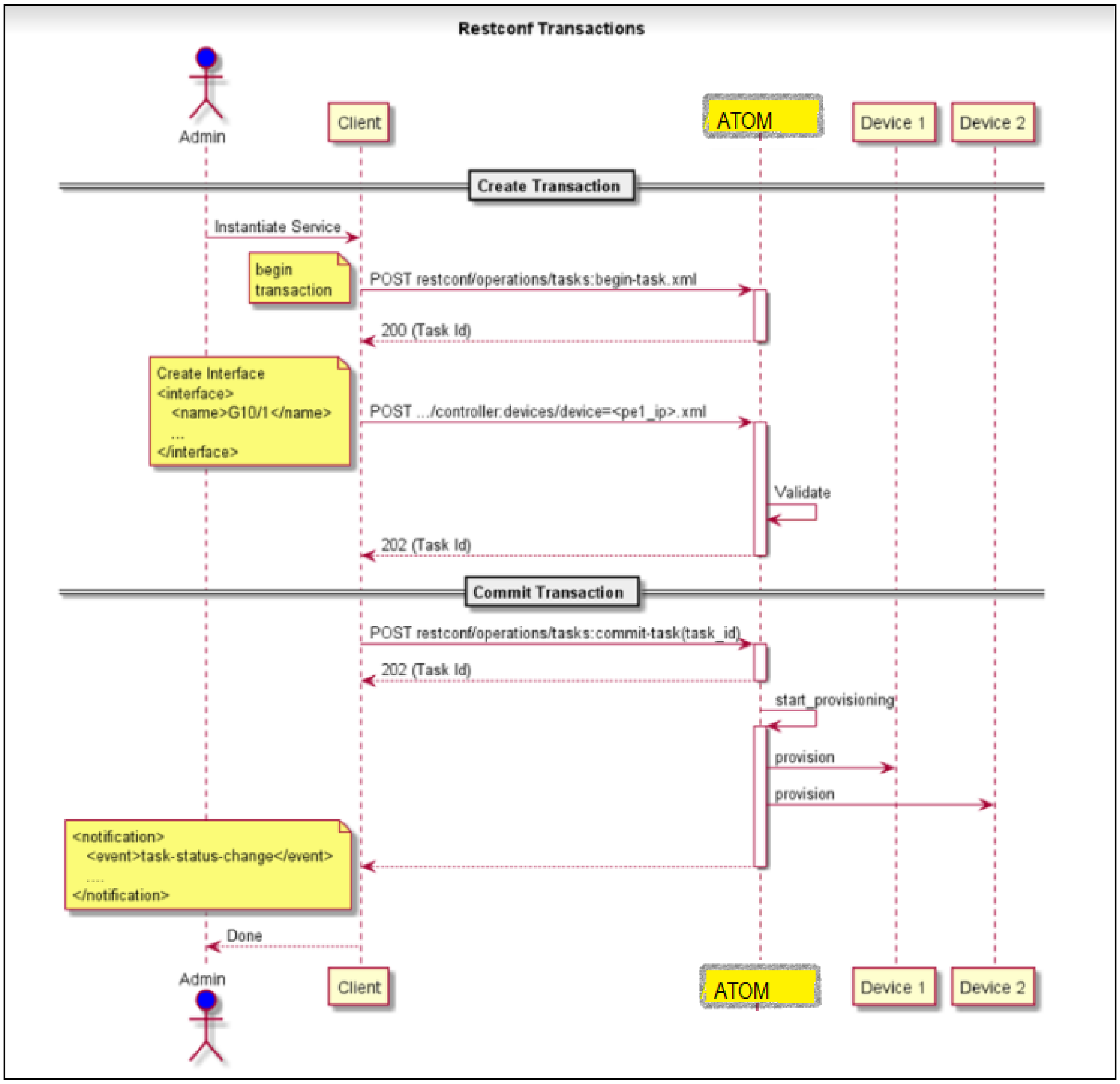
ATOM Deployment with Multiple Tenants And Sub tenancy

In this scenario , due to sub tenancy and sharing, there is potential for duplicate data unless you clarify which tenant data is being referenced. Whenever this clarity is to be provided it reflects in the API payloads (that is where you specify which tenant data you mean to use).

Transactions Support in ATOM

ATOM RESTCONF implementation supports transactions on all the data mutations and remote procedure calls (rpcs, actions). All the device operations of one transaction are done as one logical atomic operation. Commit, rollback, and retry operations are supported.

A sample transaction is illustrated below:


Restconf Transactions
Admin
Client
ATOM
Device 1
Device 2
Create Transaction
Instantiate Service
begin transaction
POST restconf/operations/tasks:begin-task.xml
200 (Task Id)
Create Interface <interface> <name>G10/1</name> ... </interface>
POST .../controller:devices/device=<pe1_ip>.xml
Validate
202 (Task Id)
Commit Transaction
POST restconf/operations/tasks:commit-task(task_id)
202 (Task Id)
start_provisioning
provision
provision
<notification> <event>task-status-change</event> .... </notification>
Done

-

1. A transaction is initiated by calling the '**begin-task**' protocol operation which returns a taskID.
2. All subsequent calls carry a header '**X-TASK-ID**' with the above taskID.
3. When the client is ready to commit the transaction, it calls the '**commit-task**' operation.
   a. Progress of the task can be polled or notifications can be subscribed.
   b. In case of any error, the commit can be retried by calling **'commit-task'** again.
   c. To rollback, the client calls '**rollback-task**'.

   ATOM computes compensating commands for devices to which commands were sent in the transaction previously and applies the compensating commands to rollback. If any errors occur during the rollback, rollback can be retried calling 'rollback-task' again.

   d. If the rollback is unsuccessful, manual intervention may be required.

4. Clients can monitor the Task detail to learn the progress of provisioning including any successful or failed device configuration attempts and commands.

#	Description	Client Request	Output	Notes
1	A transaction is started by invoking 'begin-task' rpc	POST `/restconf/operations/tasks:begin-task`	<taskID>id</taskID> **Example** `<taskID>6db7bfb2-930f-4257-98a7-7b01dd342e04</taskID>`	This taskID should be used in all the subsequent operations that are required for this transaction. All the subsequent calls should include a header. X-TASK-ID = taskID **Example** `X-TASK-ID = 6db7bfb2-930f-4257-98a7-7b01dd342e04`

2	Create a VRF	POST `/controller:devices/device=<device_id>`
		
		With header
		`X - TASK - ID = 6db7bfb2 - 930f - 4257 - 98a7 - 7b01dd342e04`
		
		Payload:
		`<vrf>`
		`<name>pevrf</name>`
		`<rd>100:1</rd>`
		`</vrf>`
			Status:
			HTTP/1.1
			202 Accepted
			
			`<taskId>6db7bfb2-930f-4257-98a7-7b01dd342e04</taskId>`	

3	Any additional CRUD operations Can be done using the same TASKId and the header.			
4	Commit the transaction	POST /restconf/operations/tasks:commit-task.xml With Header	Status: HTTP/1.1 202 Accepted `<taskId>6d`	

		X-TASK-ID = 6db7bfb2-930f-4257-98a7-7b01dd342e04	`b7bfb2-930f-4257-98a7-7b01dd342e04</taskId>`	
5	Check the task details or wait for notification	POST `restconf/operations/tasks:get-basic-task-detail.xml` Payload: `<taskId>6db7bfb2-930f-4257-98a7-7b01dd342e04</taskId>`		

Setting the transaction control options

Default transaction behavior is specified by the global transaction policy parameter and they can be overridden by per transaction policy parameters. Following parameters are available for clients to control the transaction behavior.

Option	Type	Description
do-not-send-commands-to-devices	boolean	Controls whether commands can be sent to the device. **true**: Commit the data to ATOM datastore only, but no configuration changes will be applied on the device. Useful for testing or in the case of a brownfield environment to create services. **NOTE**: This flag will be effective only if global dry-run flag is set to false
validation-scope-type 1. COMMITTED_DATA	enum	Controls whether data validation scope is across transactions. This flag is similar to isolation control in traditional RDBS systems, but limited to just data validation. Allowed values are "COMMITTED_DATA" and "UNCOMMITTED_DATA". Validation will be done only using the committed data. Current transaction will not see changes done by other parallel transactions.

2.UNCOMMITTED DATA		This is the default selection. Data validation will be done using the uncommitted data. Current transaction will see changes done by other parallel transactions. This facility can be useful to assess the potential for transaction success/failure (for example during an approval process).
fail-fast	boolean	flag to control if reference validation to be done on each payload submission. false will defer the validation until the commit-task call. This lets the client submit payloads in an out of order (reference wise).
command-sequence-policy	enum	Controls whether the generated commands need to be ordered according to the dependencies specified in the model. Allowed values are DEPENDENCY_BASED and NONE

Understanding the Task States

Each operation (create, delete or update) on any entity managed by ATOM generates a task in ATOM.

A task can be in different states of progress and each state is described in the table below:

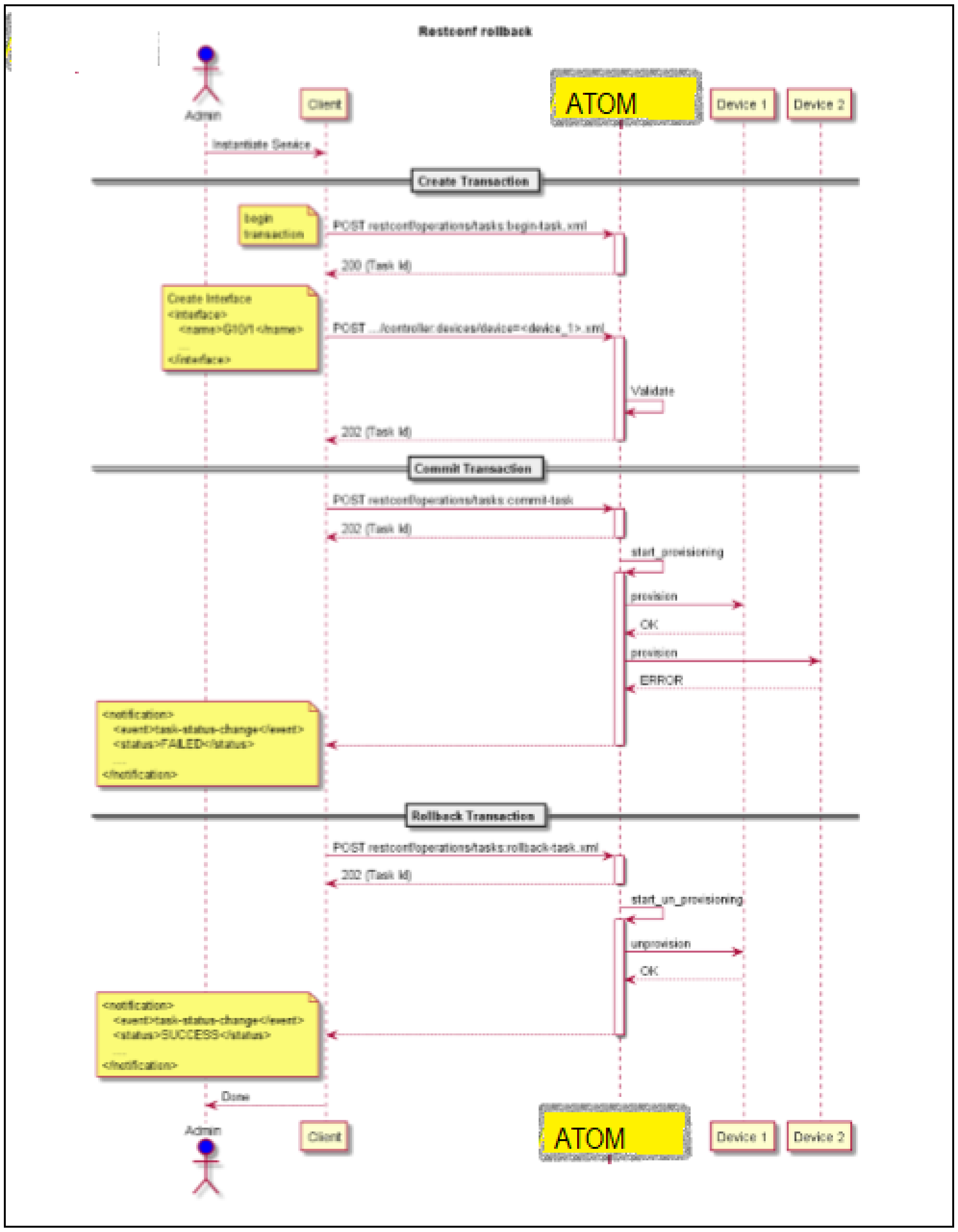
Task States	% Completion	Task Status	Description
Yet to Begin	0%	NOT_STARTED	Execution not started
Running States	0% to 100%	IN_PROGRESS	
		VALIDATED	
		RESOURCES_RESERVED	
		RESOURCES _PROVISIONED	
		OPERATIONAL_RESOURCES_RESERVED	Resources allocated for service complete on ATOM
		OPERATIONAL_RESOURCES_UNRESERVED	Service allocated resources are unreserved on ATOM
	50%	SCHEDULED_FOR_PROVISION	Task is scheduled for provisioning

	50%	WAITING	Task is waiting for approval to provision configuration
END STATES			
SUCCESS		COMPLETE	Operation completed successfully
ERROR		RESERVE_RESOURCES_FAILED	
		RESERVE_OPERATIONAL_RESOURCES_FAILED	Resource allocation (config generation) for service is failed on ATOM
		PROVISION_RESOURCES_FAILED	Execution of configuration of service on the device failed or device connection timeout
		ERROR	Some of the error messages: ● Transaction rolled back. Rollback completed. ● Operation failed on agent: ● Database exception ● Internal exception ● Operation failed due to device connectivity issues during service provisioning

Handling Transaction Failures

If the transaction commit fails, the client can retry the commit by invoking the ‘commit-task’ operation again.

ATOM will apply only those commands that are not provisioned on the devices yet. If the client wishes to roll back the transaction, he can invoke ‘rollback-task’ operation. ATOM undoes the provisioning done prior to the failure, thus restoring the device configuration to the original state. If the rollback also fails, the client retrieves the command set that needs to be applied on the device to restore the device configuration to its original state. The retrieved command set can manually be applied. Rollback can be retried by calling ‘rollback-task’ operation again.



Using Query Parameters

Each RESTCONF operation allows zero or more query parameters to be present in the request URI. The specific parameters that are allowed depends on the resource type, and sometimes the specific target resource used, in the request.

Fields Query Parameter

The "fields" query parameter is used to optionally identify data nodes within the target resource to be retrieved in a GET method. The client can use this parameter to retrieve a subset of all nodes in a resource.

Example - Retrieving all the entries of the target container

Let us assume that the target source is a Credential Set. To retrieve only the “`name`” and the “`transport-type`” fields of all the Credential Sets contained within ATOM, execute the following query:

```
/restconf/data/controller:credentials/credential-sets?fields=credential-set(na
me;transport-type)
```

Example - Retrieving a specific entry of the container

Let us assume that the target source is a particular credential set of name “`xyz`”. To retrieve only the “`name`” and the “`port number`” fields of this credential set only, execute the following query:

```
/restconf/data/controller:credentials/credential-sets/credential-set=xyz?field
s=name;port-number
```

Example - Retrieving the child node of a node by providing path in the fields

Let us assume that the target source is “devices”. To retrieve the “`name`” and the “`rd`” fields of the “`vrf`” , a child node of the “`device`”:

```
/restconf/data/controller:devices.xml?fields=device/l3features:vrfs/vrf(name;r
d)
```

Depth Query Parameter

The "depth" query parameter is an optional parameter which can be used to retrieve all the nodes under a resource till a specified hierarchy tree depth.

Example - Retrieving all the entries of the target device for a specified depth

Let us assume that the target device is 172.16.1.130. Instead of retrieving complete tree under a device which can be of unknown depth, to retrieve only the interested childs available till a depth of 2 contained within ATOM, execute the following query:

```
/restconf/data/controller:devices/device=172.16.1.130.xml?depth=2
```

Pagination in APIs

While sending an API request to ATOM, you can specify the number of resources, and the specify the record, starting from which the entries should be returned in the response body. The request header should contain the “limit” and the “offset” keywords in the URL.

Using the Limit parameter

The limit parameter tells the API how many records should be retrieved from the entire set of results:

```
/restconf/data/controller:credentials/credential-sets.xml?limit=2
```

Using the Offset parameter

The “offset” parameter tells the API where to start returning records from the entire set of results.

Let us assume that there are four Credential Sets available in the following order:

- Set 1
- Set 2
- Set 3
- Set 4

```
Request: GET
/restconf/data/controller:credentials/credential-sets.xml?limit=2&offset=1

Response:
       Set 2
       Set 3
```

```
Request:
/restconf/data/controller:credentials/credential-sets.xml?limit=2&offset=2

Response:
Set 3
Set 4
```

```
Request: /restconf/data/controller:credentials/credential-sets.xml?offset=3

Response:
Set 4
```

Invoking ATOM APIs from REST Client

Introduction

You can access the ATOM RESTCONF APIs using either the GUI (REST Client) or CLI methods (cURL). Clients use HTTP methods such as GET, POST, PUT, and DELETE to make Requests to the ATOM server. The base URL to perform a RESTCONF based operation is https://<server-ip>/restconf, where <server-ip> is the IP address of the ATOM VM

Any client application that can send HTTP Requests over a secure channel by using SSL can be an appropriate tool for developing RESTful applications with the ATOM API.

Constructing the API Request

A typical RESTCONF request is outlined below:

For this example, the web browser is Google Chrome and the REST client is POSTMAN and the chosen REST operation is GET.

1. Login to the VM using the administrator's credentials
2. Open the POSTMAN rest client tool and follow the steps as outlined below:
3. In the **Normal** tab, go to the **Enter request URL** field, enter the URL in the following format.

   Base URL : https://<server-IP>/restconf/data/controller.xml, where *server-IP* is the IP address of the ATOM server VM

4. Click the **Authorization** tab, choose **Bearer Auth** as the Authentication type.
5. Click the **Headers** tab to enter the following information:

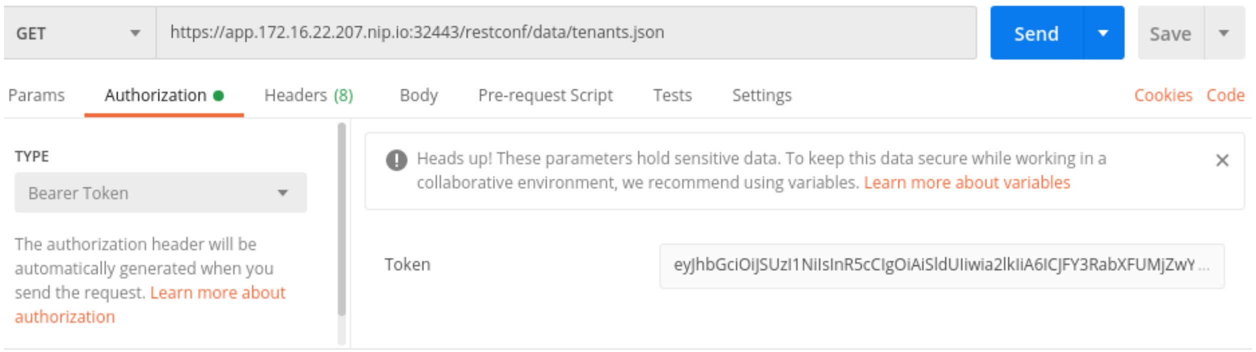
   Enter **Content-Type** and value as application/xml.

6. Select the appropriate operation (any of the RESTCONF operations) to be performed on the API object.
7. Click **Send**
8. The resulting Response pattern is displayed in the text box.

Example

You can retrieve all the child entities in Credentials as illustrated below:

1. Open the POSTMAN client session in another browser.
2. In the **Authorization** tab, enter the token obtained as specified in the Authentication.

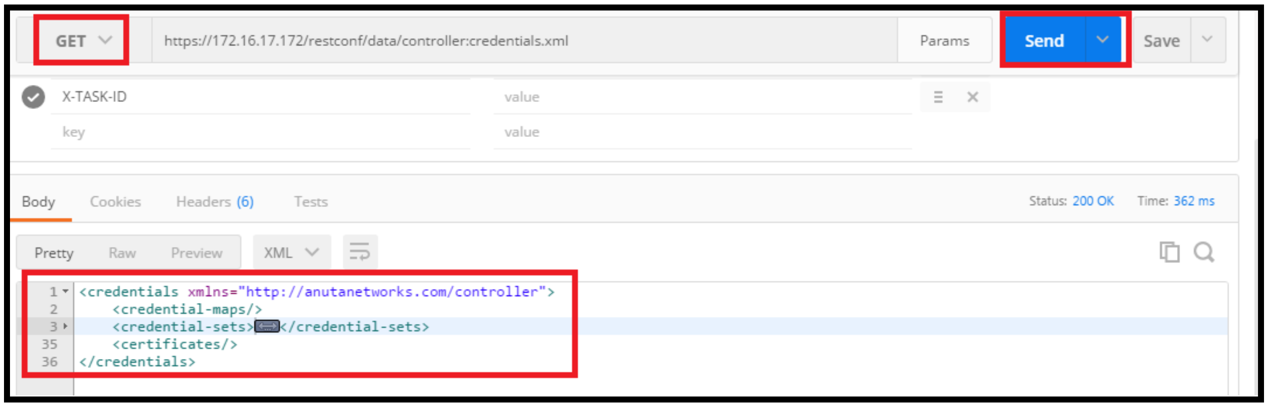


The authorization header will be automatically generated when you send the request. Learn more about authorization

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. Learn more about variables

Token: eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJFY3RabXFUMjZwY...

3. Click **Send** to send the request to the ATOM server.
4. Click the **Body** tab to see the Response obtained for the Request



```
<credentials xmlns="http://anutanetworks.com/controller">
    <credential-maps/>
    <credential-sets>⋯</credential-sets>
    <certificates/>
</credentials>
```

Generating the X-TASK-ID

Before performing any RESTCONF operation such as POST, PUT or DELETE, a transaction ID should be generated. This transaction ID or the task ID is used to poll ATOM for all the subsequent tasks; **commit_task and get-details-task**. For all the RESTCONF operations, except GET, follow these steps:

1. Begin a transaction.

```
Request
URI https://<VM-IP>/restconf/operations/tasks:begin-task.xml
where VM-IP is the IP address of the ATOM VM
Payload: Not Applicable

Operation: POST
```

```
Headers
      APIVersion: 2.0
      Content-Type: application/xml

Response
      200 OK
      <taskId>613f5f45-dd29-4783-8527-7303756db312</taskId>
```

2. Enter the value of the <`taskId`>, generated in the above step, in the X-TASK-ID field in the Headers tab.

3. Commit the transaction.

```
Request
URI https://<VM-IP>/restconf/operations/tasks:commit-task.xml
where VM IP is the IP address of the ATOM VM

Payload: Not Applicable

Operation: POST

Headers
      APIVersion: 2.0
      Content-Type: application/xml
      X-TASK-ID: 613f5f45-dd29-4783-8527-7303756db312

Response
      200 OK
      <taskId>613f5f45-dd29-4783-8527-7303756db312</taskId>
```

The task can be monitored by the following: https://<VM-IP>/restconf/operations/tasks:get-full-task-details.xml. Track task details by using above URL at that instance.

```
Request
URI https://<VM-IP>/restconf/operations/tasks:get-full-task-detail.xml
where VM IP is the IP address of the ATOM VM

Payload:<taskId>613f5f45-dd29-4783-8527-7303756db312</taskId>

Operation: POST

Headers
      APIVersion: 2.0
      Content-Type: application/xml

Response
      200 OK
```

Using the POST operation on the URL, you can periodically check progress until the task is completed.

Generating the Payload

From the "Schema Browser", obtain the path of the entity which needs to be accessed. This will give the XML schema file of the YANG entity defined in ATOM.

Fill in the values for the fields described in the schema and send the request to ATOM as described in the section "Constructing the API Request "

Examples of ATOM APIs

This section contains different examples of restconf APIs for respective ATOM entity involved.

Adding a Credential Set

```
Request
URI https://<VM-IP>/restconf/data/controller:credentials/
credential-sets.xml?
where VM_IP is the IP address of the ATOM VM
Body:
<credential-set>
      <enable-password>
      elastic
      </enable-password>
      <snmp-version>
      SNMPV2C
      </snmp-version>
      <cli-configcmd-time-out>
      100
      </cli-configcmd-time-out>
      <time-out>
      10
      </time-out>
      <username>
      admin
      </username>
      <port-number>
      23
      </port-number>
      <snmp-read-community-str>
      public
      </snmp-read-community-str>
      <config-retrieval-credential>
      true
      </config-retrieval-credential>
      <password>
      Elastic+123
      </password>
      <transport-type>
      TELNET
      </transport-type>
```

```
        <name>
        restIOS
        </name>
        <cr-wait-time>
        10
        </cr-wait-time>
        <command-execution-wait-time>
        500
        </command-execution-wait-time>
        <number-of-retries>
        10
        </number-of-retries>
        <cr-time-out>
        5
        </cr-time-out>
</credential-set>

Response
        Status:202
        Reason: Accepted
```

Onboarding a device to ATOM

```
Request
URI https://<VM-IP>/restconf/data/controller:devices.xml?
where VM_IP is the IP address of the ATOM VM

Body:
        <device>
                <name>
                device139
                </name>
                <management-mode>
                MANAGED
                </management-mode>
                <manage-by-management-station>
                false
                </manage-by-management-station>
                <credential-set>
                restIOS
                </credential-set>
                <id>
                139device
                </id>
                <mgmt-ip-address>
                172.16.1.139
                </mgmt-ip-address>
        </device>

Response
        Status:202
        Reason: Accepted
```

Creating a vrf entry on the device

```
Request
URI
https://<VM-IP>/restconf/data/controller:devices/device=<device-id>/vrfs.xm
l?
where VM_IP is the IP address of the ATOM VM
where device-id is the unique ID with which device is onboarded to ATOM VM

Body:
      <vrf>
            <name>
            vrf-internal
            </name>
      </vrf>

Response
      Status:202
      Reason: Accepted
```

Creating the Service

```
Request
URI https://<VM-IP>/restconf/data/controller:services.xml?
where VM_IP is the IP address of the ATOM VM

Body:
      <l3service>
            <name>test_sub_intf2</name>
            <device-ip>172.16.1.139</device-ip>
            <interface-mode>sub-interface</interface-mode>
            <vrf>vrf2</vrf>
            <vlan-id>1002</vlan-id>
            <interface>FastEthernet8</interface>
            <ip-address>192.168.15.1</ip-address>
            <netmask>255.255.255.0</netmask>
      </l3service>

Response
      Status:202
      Reason: Accepted
```

Webhook API Support

Users can invoke Workflows defined in ATOM using webhook url.

Webhook URL :

curl -X POST -k -d <payload> -H "Content-Type: application/json" "http://username:password@<**server_ip**>/rest/webhook?**wf_name**=<workflowName>&**atom_user_name**=<userName>&**atom_user_owner**=<owner>"

Wf_name : workflow name**.**
Atom_user_name: atom user name
Atom_user_owner: tenant name

Query params wf_name, atom_user_name and atom_user_owner are mandatory params.

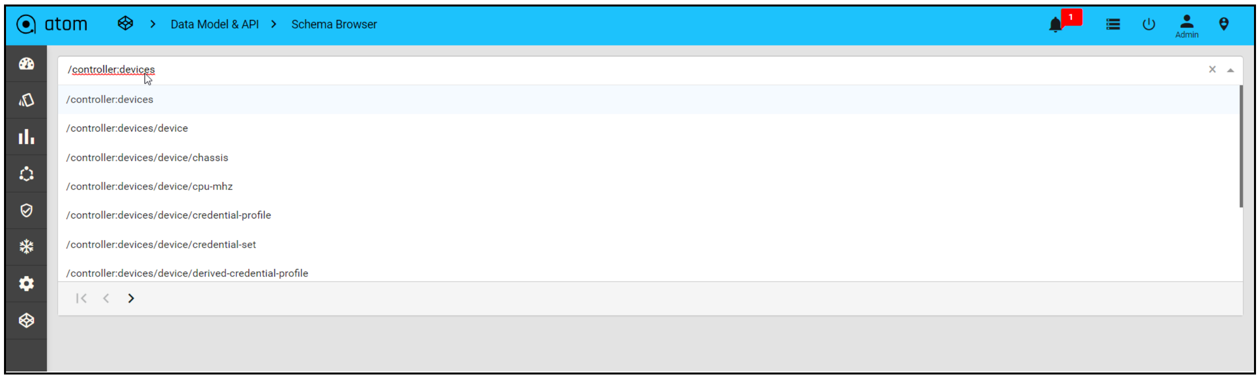
Example :
curl -X POST -k -d '{"param1":"test","param2":"test2", "var1":[1,3,4,5]}' -H "Content-Type: application/json" "http://username:password@<**server_ip**>/rest/webhook?**wf_name**=createDevice&**atom_user_name**=systemuser&**atom_user_owner**=system"

Tools for API Development and Testing

Schema Browser

Schema Browser is a built-in tool of ATOM that provides a graphical view of the data model objects maintained in ATOM. You can query for the required object by providing the instance path of the target.

1. Login to ATOM and navigate to **Administration > System > General Settings > Developer Options**
2. Select the **Enable Developer Mode** option.
3. After selecting this option, navigate to **Developers Tools > Schema Browser**
4. In the search field, enter the path of the target to know the YANG schema of the object
5. For example, to retrieve the schema file of the Credential Set object as shown below:
    - Enter the path of the target in the Schema Path box.
    - Select the path from the drop-down menu.



/controller:devices
/controller:devices
/controller:devices/device
/controller:devices/device/chassis
/controller:devices/device/cpu-mhz
/controller:devices/device/credential-profile
/controller:devices/device/credential-set
/controller:devices/device/derived-credential-profile

The schema file of the object defined in the data model is displayed in the below pane.

Looking into the schema representation, you can now construct the payload that can be sent in the Request body of an ATOM API. This can be sent as an input in the Request Body while constructing an API query.

Swagger

You can now leverage the rich API set of ATOM using the Swagger UI integrated with ATOM.

1. Enable the Developer Mode in ATOM as follows:

   Navigate to **Administration > System > General Settings > Enable-Developer-Mode**
2. Click **RestConf APIs** option by navigating to **Developer Tools > RestConf APIs**
3. In the newly opened browser tab of APIs Documentation, **Select API** drop-down, select the entity modeled in ATOM and you would like to explore the corresponding API request, response patterns.
   - **RESTCONF Data**

     Select this option to view the APIs of the configuration data of the end points modeled in YANG
   - **RESTCONF RPC**

     Select this option to view all the APIs of the RPCs modeled in YANG
   - **RESTCONF Services**

     To view the APIs required to construct the service models in YANG
4. In the selected entity, look at the Request parameters that are required to construct the payload and substitute the values in these parameters by using the "**Try it out**" option available in the Swagger UI.

Scenarios

1. Automation Activity
   a. Connecting to ATOM

   b. Submitting a Task to ATOM
   c. Polling Task Status from ATOM
   d. Asynchronous Notification from ATOM
2. Receive Notifications/Alerts on Slack/Email etc. from ATOM
3. Trap Forwarding from ATOM to External Client
4. Client receiving Task updates via ATOM Streams
5. Consumer listening on ATOM Kafka Topic
6. Invoking a Workflow with Notification from ATOM File Server. (e.g: File/Image download completion from ATOM File Server to Network Device)

Notifications Support In ATOM

ATOM supports the following notification definitions

1. YANG Notifications
2. ATOM ChangeLog Notifications
3. Yang Notifications generated by Devices (when they support)
4. NAAS Events (Old generation style but still useful)
5. ATOM Infrastructure Monitoring Alerts
6. Device/Network Monitoring Alerts

Overall, the following transports are supported

1. SSE
2. Kafka Topics
3. Web Sockets
4. RabbitMQ

Notification Type And Detail

Notification Type	Notes
YANG Notification	These are the notifications defined in Yang modules and Sub modules, according to YANG 1.1 spec. Notifications defined in ATOM yang modules will be implemented by ATOM. If a user adds their own Yang schema with Notifications, the implementation details need to be supplied along with the Yang Schema. The way to supply the implementation is

	by defining an 'ATOM Change Log' script. See the next item.
ATOM Change-Log Notification (ACLN)	Custom Implementation of ATOM. Although Yang Notification is the prescribed way of defining notification structures, when a client defines a yang notification the logic to generate the notification payload has to be implemented by ATOM development team. It is understood that clients may not always be willing to change yang models to add new notification definitions nor they can wait for ATOM team to implement them. For those scenarios, clients can use the ACLN way of adding notifications. ACLN lets the client 1. To define condition to be matched against the transaction change log 2. To refer to a yang notification or specify a structure/payload of the notification 3. To provide logic to generate the notification ATOM provides access to the change log and utilities. Using this facility, it is possible to extend the notification capabilities of ATOM dynamically.
YANG Notifications generated by Devices	As specified in YANG 1.1 Spec and Restconf Spec. Users can configure ATOM to subscribe for Netconf Notifications generated by devices. On top of it, users can also choose to expose those notifications to the North bound clients via SSE transport.
NAAS Events	Custom Implementation of ATOM. NAAS Event is an old generation style notification but it is still useful.
ATOM Infra monitoring Alerts	Custom Implementation of ATOM. ATOM infra components such as ATOM services, Infra services etc are monitored and alerted when some conditions are met. These alerts can be acted upon in various ways such as emailing, slack channels, invoke workflows, raise tickets etc. And these are published to a

	kafka topic, so integrators can write consumer logic against these.
Device Monitoring Alerts	Custom Implementation of ATOM If customers enable the Device Telemetry component of their ATOM deployment, devices are monitored and alerted when some conditions are met. These alerts can be acted upon in various ways such as emailing, slack channels, invoke workflows, raise tickets etc. And these are published to a kafka topic, so integrators can write consumer logic against these.

Notification Type, Payload Schema Information

Notification Type	Brief Notes	Schema Type
YANG Notification	As Specified in YANG 1.1 Spec and Restconf Spec	As per YANG 1.1 spec
ATOM Change-Log Notification	Custom Implementation of ATOM	The data structure of the notification itself is as specified by the user. If the notification definition is referring to a yang notification, then, the schema of the payload is driven by yang definition. Otherwise, there is no formal schema for the notification payload. Since this structure is defined by the user (not ATOM platform), they know what to expect as part of the payload.
YANG Notification generated by Devices	As Specified in YANG 1.1 Spec and Restconf Spec	As per YANG 1.1 spec
NAAS Events	Custom Implementation of ATOM	Overall structure is defined as a yang model.
ATOM Infra monitoring Alerts	Custom Implementation of ATOM	Alert structure is defined as a yang model.
Device Monitoring Alerts	Custom Implementation of ATOM	Alert structure is defined as a yang model.

SSE (Server Sent Events) Support

ATOM Supports SSE Transport as specified in the Restconf spec.

Fetching Stream Information

GET /restconf/data/ietf-restconf-monitoring:restconf-state/streams

Sample Output

```
<streams xmlns="urn:ietf:params:xml:ns:yang:ietf-restconf-monitoring">
    <stream>
      <name>NETCONF</name>
      <description>default NETCONF event stream. There are other purpose built streams
available.</description>
      <replay-support>true</replay-support>
      <access>
        <encoding>xml</encoding>
        <location>/streams/NETCONF</location>
      </access>
      <access>
        <encoding>json</encoding>
        <location>/streams/NETCONF/json</location>
      </access>
    </stream>

    <stream>
      <name>ACLN</name>
      <description>event stream to receive  changeLog notifications from ATOM.</description>
      <replay-support>true</replay-support>
      <access>
        <encoding>xml</encoding>
        <location>/streams/ACLN</location>
      </access>
      <access>
        <encoding>json</encoding>
        <location>/streams/ACLN/json</location>
      </access>
    </stream>

    <stream>
      <name>TASKS</name>
      <description>event stream to receive Task updates from ATOM.</description>
      <replay-support>true</replay-support>
      <access>
        <encoding>xml</encoding>
        <location>/streams/TASKS</location>
      </access>
      <access>
        <encoding>json</encoding>
        <location>/streams/TASKS/json</location>
      </access>
    </stream>

    <stream>
      <name>ALARMS</name>
      <description>event stream to receive ALARM notification from ATOM.</description>
      <replay-support>true</replay-support>
      <access>
        <encoding>xml</encoding>
        <location>/streams/ALARMS</location>
      </access>
      <access>
        <encoding>json</encoding>
        <location>/streams/ALARMS/json</location>
      </access>
    </stream>
  </streams>
```

List of Streams (for a quick glance)

Extracted from the above xml
/streams/NETCONF
/streams/TASKS
/streams/ALARMS
/streams/ACLN

Sample SSE Session

```
curl -u admin:admin localhost:8080/restconf/streams/stream/NETCONF

data:<notification
xmlns="urn:ietf:params:netconf:capability:notification:1.0"><eventTime>2020-01-05T12:33:58.078Z</eventTime
><task-id>Fge1AvVq1ORg-VJaWwGKj47A</task-id><report>   <notification-spec>$$tasks$$</notification-spec>
<anchor-object></anchor-object>   <source-datanode>Fge1AvVq1ORg-VJaWwGKj47A</source-datanode>
<source-datanode-owner>system</source-datanode-owner>   <matcher-type>DEFAULT</matcher-type>
<generator-type>DEFAULT</generator-type>
<timestamp>2020-01-05T12:33:58.078Z</timestamp></report><task-notification><id>Fge1AvVq1ORg-VJaWwGK
j47A</id><status>IN_PROGRESS</status><operation-name><![CDATA[Create:
vendor]]></operation-name><starttime>1578227638075</starttime><endtime>0</endtime><message>null</m
essage></task-notification></notification>
```

Sample SSE Python code

Use the below client script to receive notification from ATOM

Usage :

To receive notifications using user credentials

1. python3 notification.py -k "172.16.18.153" -upriv "admin" -kp "443" -c “atom” -u "admin" -mp "Secret@123" -s "TASKS"

Copy client secret from response

2. python3 notification.py -k "172.16.18.153" -upriv "user" -kp "443" -r "system" -c "atom" -u "john" -p "Secret@123" -s "TASKS" -cs "dc705bda-41c7-459e-9fa2-306567624d11"

To receive notifications using admin credentials

1. python3 notification.py -k "172.16.18.153" -upriv adminstream -kp "443" -r "system" -c "atom" -u "admin" -mp "Secret@123" -p "Secret+123" -s "TASKS"

To receive notifications in json format use: -f "json"

Note: below block of code should be saved with name notification.py

```
import requests
import urllib3
import argparse
import ast
import warnings
import time
warnings.filterwarnings("ignore")
urllib3.disable_warnings()


class KeycloakManager:

      def __init__(self, keycloak_host, keycloak_port):
          self.realm = "system"
          self.client = "atom"
          self.user = ""
          self.password = ""
          self.master_password = ""
          self.stream_format = ""
          self.keycloak_host = keycloak_host
          self.keycloak_port = keycloak_port
          if keycloak_port == "443":
              self.keycloak_url = "https://" + self.keycloak_host
          else:
              self.keycloak_url =
"https://app."+self.keycloak_host+".nip.io:" + self.keycloak_port
          self.master_auth_openid_token_url = self.keycloak_url +
"/auth/realms/master/protocol/openid-connect/token"

      def check_keycloak_status(self):
          trials = 20
          i = 0
          while i < trials:
              try:
```

```
            x = requests.get(self.keycloak_url, verify=False)
            if x.status_code == 200 or x.status_code < 300:
                  return 0
            else:
                  print("Status code: " + str(x.status_code))
                        time.sleep(20)
                        i = i + 1
              except Exception as e:
            print(e)
            time.sleep(20)
            i = i + 1
          return 1

      def get_token(self):
          print("Fetching token from client {} ".format(self.client))
          params = {'client_id': self.client, 'grant_type': 'password',
'username': self.user, 'password': self.password}
          x = requests.post(self.master_auth_openid_token_url, params,
verify=False).content.decode('utf-8')
          time.sleep(1)
          # print ('\n')
          return ast.literal_eval(x)['access_token']

      def get_client_secret(self, token):
          client_url = self.keycloak_url + "/auth/admin/realms/" + self.realm
+ "/clients/?clientId=" + self.client
          headers = {
            'content-type': 'application/json',
            'Authorization': 'Bearer ' + token
          }
          x = requests.get(client_url, verify=False, headers=headers).json()
          time.sleep(1)
          id = x[0]['id']

          #/{realm}/clients/{id}/client-secret
          client_url = self.keycloak_url + "/auth/admin/realms/" + self.realm
+ "/clients/" + id + "/client-secret"
          headers = {
              'content-type': 'application/json',
              'Authorization': 'Bearer ' + token
          }
          x = requests.get(client_url, verify=False, headers=headers).json()
          time.sleep(1)
          return x['value']

      def get_access_token(self, client_secret):
          request_url = self.keycloak_url+"/auth/realms/" + self.realm +
"/protocol/openid-connect/token"
          params = {'client_id': self.client,'grant_type': 'password',
'client_secret': ''+client_secret+'', 'scope' :'openid', 'username':self.user,
'password': self.password}
          print(params)
          x = requests.post(request_url, data = params,
verify=False).content.decode('utf-8')
          time.sleep(1)
          print(x)
```

```
            return ast.literal_eval(x)['id_token']

        def notification(self, id_token, stream):
            headers = {'Authorization': 'Bearer ' + id_token}
            print (headers)
            url = ''
            if(self.stream_format == 'xml'):
                url = self.keycloak_url+"/restconf/streams/stream/"+stream
                print (url)
            elif(self.stream_format == 'json'):
                url =
self.keycloak_url+"/restconf/streams/stream/"+stream+"/json"
                print (url)
            with requests.get(url, stream=True, verify=False, headers=headers,
timeout=None) as r:
                for line in r.iter_lines(chunk_size=1):
                print (line)

if __name__ == '__main__':
      parser = argparse.ArgumentParser()
      parser.add_argument('-k', '--keycloak-server', dest='keycloak_server',
help='keycloak host')
      parser.add_argument('-kp', '--keycloak-port', dest='keycloak_port',
help='keycloak port')
      parser.add_argument('-r', '--realm', dest='realm', default='system',
help='keycloak realm')
      parser.add_argument('-c', '--client', dest='client', default='atom',
help='keycloak client')
      parser.add_argument('-upriv', '--user-privilege',
dest='upriv',choices=['admin', 'user', 'adminstream'], help='user privilege
admin/user/adminstream')
      parser.add_argument('-u', '--user', dest='user', default='admin',
help='keycloak admin user')
      parser.add_argument('-p', '--password', dest='password',
default='admin', help='keycloak admin password')
      parser.add_argument('-mp', '--master-password', dest='master_password',
help='keycloak master admin password')
      parser.add_argument('-s', '--stream', dest='stream', choices=['TASKS',
'ALARMS', 'ACLN', 'NETCONF'], help='stream of the notification')
      parser.add_argument('-cs', '--client-secret', dest='client_secret',
help='client secret')
      parser.add_argument('-f', '--format', dest='stream_format',
choices=['json', 'xml'], default='xml', help='format xml/ json')

      args = parser.parse_args()

      keycloak_server = args.keycloak_server
      keycloak_port = args.keycloak_port
      realm = args.realm
      user = args.user
      password = args.password
      client = args.client
      stream = args.stream
      upriv = args.upriv
      assert upriv in ['admin', 'user', 'adminstream'], 'user privilege must
be admin, user or adminstream'
```

```
    stream_format = args.stream_format
    keycloak_object = KeycloakManager(keycloak_server, keycloak_port)

    keycloak_object.user = user
    keycloak_object.realm = realm
    keycloak_object.client = "admin-cli"
    keycloak_object.stream_format = stream_format
    if keycloak_object.check_keycloak_status() == 0:
          print("Keycloak is active")
    pass
    else:
    print("Keycloak is inactive. Exiting..")
    exit(1)

    if (upriv =="admin"):
    master_password = args.master_password
    keycloak_object.password = master_password
    token = keycloak_object.get_token()
    keycloak_object.client = client
    client_secret = keycloak_object.get_client_secret(token)
    print("clientSecret:" + client_secret)
    elif(upriv =="user"):
    client_secret = args.client_secret
          keycloak_object.client = client
    keycloak_object.password = password
    id_token = keycloak_object.get_access_token(client_secret)
    print("id_token:" + id_token)
    keycloak_object.notification(id_token,stream)
    elif(upriv =="adminstream"):
    master_password = args.master_password
    keycloak_object.password = master_password
    token = keycloak_object.get_token()
    keycloak_object.client = client
    keycloak_object.password = password
          client_secret = keycloak_object.get_client_secret(token)
    print("clientSecret:" + client_secret)
    id_token = keycloak_object.get_access_token(client_secret)
    print("id_token:" + id_token)
    keycloak_object.notification(id_token,stream)
```

ATOM Change Log Reactions

ACLR provides facilities to listen for transaction changes and express reactions to the change.

The reaction can be

1. **Raise a Notification** (We covered this aspect as ATOM Change Log Notification, ACLN)
2. **Invoke some code** (That will run on its own transaction)

3. **Veto the change** (if you need to rollback an on-going transaction based on a condition that is not already modeled)

At a high level, ACLR has the following structure

```
reaction-definitions {
  definition {
    Name;
    context {
      anchor-object;
      change-type;
    }
    trigger-condition {
      // you can use xpath expression , groovy script etc to code the condition
    }

    reaction {
      case notification {
        notification-object-structure {
            case yang-notification {
                yang-notification-identifier;
            }
            case custom {
                notification-structure; // xml/json structure, for ex
            }
        }
      }

      case veto {
         message;
      }

      case take-action {
        action-spec // you can call a rpc,action or call a script, workflow etc.
      }
    }
  }
}
```

ATOM System Health & Availability

ATOM System Health, Availability, Performance of various components can be performed using various mechanisms:

1. Graphical User Interface - This is documented in "ATOM Deployment Guide"
    a. System Alerts can be viewed ATOM End User Interface
    b. Advanced Metrics can be viewed using Dashboards in Grafana
2. Notifications - Documented in Sections Below
3. Polling
    a. SNMP
    b. HTTP Probe

Notifications

Prometheus Alert Manager will send the notifications about each component's health and availability whenever the anomalies are found. It is a push mechanism.

To get the alerts in real time, admin has to configure the notification routing and receivers. By Default, ATOM deployment comes with slack and webhook as notification receivers.

To notify alerts to the Slack channel, the user has to provide the slack_api_url. Refer to ***Sending messages using Incoming Webhooks*** section in ***https://api.slack.com/messaging/webhooks*** to get the unique URL.

The push notifications to any http endpoint, user can add the url under webhook_configs section.

Prometheus Alert Manager can be integrated with the following receivers.

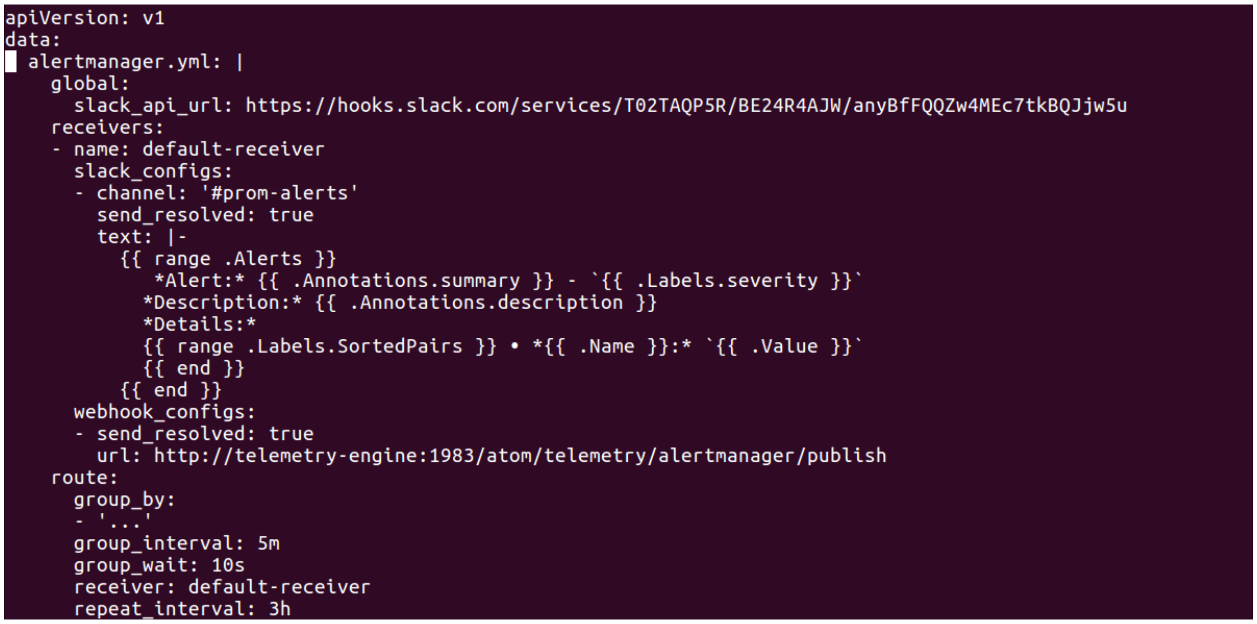
1. Slack
2. Email
3. PagerDuty
4. Pushover
5. OpsGenie
6. Webhook (It accepts any generic http endpoint)
7. WeChat

Refer to the below document for more details.

https://prometheus.io/docs/alerting/latest/configuration/#webhook_config

Use k8s dashboard (*https://<URL>/k8s/*) or login to the k8s master node and execute the below command to change the configuration.

kubectl edit cm infra-tsdb-monitoring-alertmanager -n atom



```
apiVersion: v1
data:
  alertmanager.yml: |
    global:
      slack_api_url: https://hooks.slack.com/services/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
    receivers:
    - name: default-receiver
      slack_configs:
      - channel: '#prom-alerts'
        send_resolved: true
        text: |-
          {{ range .Alerts }}
             *Alert:* {{ .Annotations.summary }} - `{{ .Labels.severity }}`
            *Description:* {{ .Annotations.description }}
            *Details:*
            {{ range .Labels.SortedPairs }} • *{{ .Name }}:* `{{ .Value }}`
            {{ end }}
          {{ end }}
      webhook_configs:
      - send_resolved: true
        url: http://telemetry-engine:1983/atom/telemetry/alertmanager/publish
    route:
      group_by:
      - '...'
      group_interval: 5m
      group_wait: 10s
      receiver: default-receiver
      repeat_interval: 3h
```

Polling

HTTP Probe

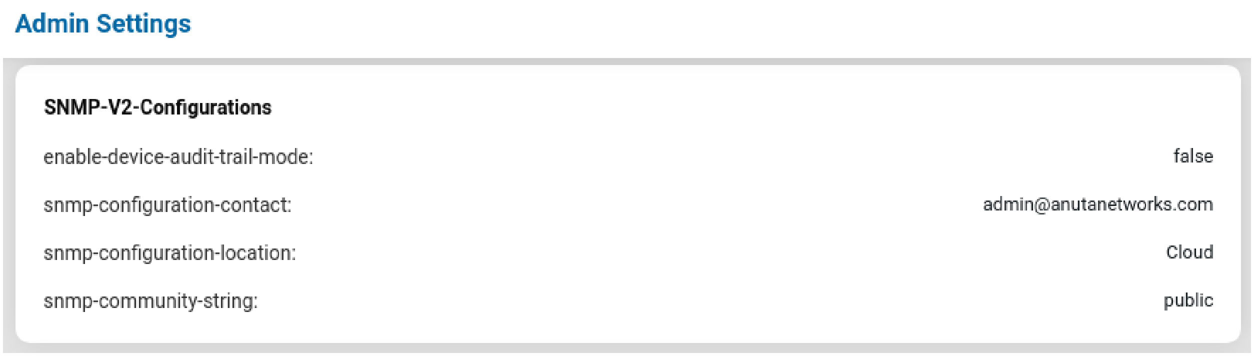
ATOM provides a way to check availability using REST Endpoint. Bearer authentication token as specified in Authentication needs to be provided

GET https://<ATOM_URL>:30443/rest/ui/systemInfo and check for the HTTP status code in response. Status code 200 will indicate that the ATOM is available.

SNMP

ATOM supports SNMP Agent and can be used by OSS Tools to perform Basic Health checks. ATOM Support SNMP Protocol Version SNMPV2c.

SNMP Access can be enabled in Admin settings from ATOM User Interface - Administration -> System -> General Settings.


Admin Settings

SNMP-V2-Configurations	
enable-device-audit-trail-mode:	false
snmp-configuration-contact:	admin@anutanetworks.com
snmp-configuration-location:	Cloud
snmp-community-string:	public

Supported MIBs:

1. SYSTEM-MIB
   a. Supported OIDs:
      i. sysDescr
      ii. sysObjectID
      iii. sysUpTime
      iv. sysContact
      v. sysName
      vi. sysLocation
      vii. sysServices
      viii. sysORLastChange
   b. Sample output for a snmpwalk on the system mib:

      # snmpwalk -v2c -c public <atom-master-ip>:port .1.3.6.1.2.1.1

```
snmpwalk -v2c -c public 172.16.22.92:30954 .1.3.6.1.2.1.1
SNMPv2-MIB::sysDescr.0 = STRING: ATOM, version: 11.9.0.0
SNMPv2-MIB::sysObjectID.0 = OID: SNMPv2-SMI::enterprises.42177.1.1
DISMAN-EVENT-MIB::sysUpTimeInstance = Timeticks: (2486102) 6:54:21.02
SNMPv2-MIB::sysContact.0 = STRING: admin@anutanetworks.com
SNMPv2-MIB::sysName.0 = STRING:
SNMPv2-MIB::sysLocation.0 = STRING: Cloud
SNMPv2-MIB::sysServices.0 = INTEGER: 72
SNMPv2-MIB::sysORLastChange.0 = Timeticks: (0) 0:00:00.00
```