Ansible Tower API Guide

Release Ansible Tower 3.3.4

Red Hat, Inc.

Jun 21, 2019

CONTENTS

1 Tools **2**

2 Browsable API **3**

3 Conventions **8**

4 Sorting **9**

5 Searching **10**

6 Filtering **11**

7 Pagination **14**

8 Access Resources **15**
- 8.1 Configuration Settings . . . 15
- 8.2 Identifier Format Protocol . . . 16

9 Read-only Fields **18**

10 Introduction to tower-cli **19**
- 10.1 License . . . 19
- 10.2 Capabilities . . . 19
- 10.3 Installation . . . 19
- 10.4 Configuration . . . 20

11 Tower API Reference Guide **24**

12 Index **25**

13 Copyright © 2019 Red Hat, Inc. **26**

Index **27**

Thank you for your interest in Red Hat Ansible Tower. Ansible Tower is a commercial offering that helps teams manage complex multi-tier deployments by adding control, knowledge, and delegation to Ansible-powered environments.

The *Ansible Tower API Guide* focuses on helping you understand the Ansible Tower API. This document has been updated to include information for the latest release of Ansible Tower 3.3.4.

We Need Feedback!

If you spot a typo in this documentation, or if you have thought of a way to make this manual better, we would love to hear from you! Please send an email to: docs@ansible.com

If you have a suggestion, try to be as specific as possible when describing it. If you have found an error, please include the manual's title, chapter number/section number, and some of the surrounding text so we can find it easily. We may not be able to respond to every message sent to us, but you can be sure that we will be reading them all!

Ansible Tower Version 3.3.4; January 23, 2019; https://access.redhat.com/

CHAPTER

ONE

TOOLS

This document offers a basic understanding of the REST API used by Ansible Tower.

REST stands for Representational State Transfer and is sometimes spelled as "ReST". It relies on a stateless, client-server, and cacheable communications protocol, usually the HTTP protocol.

You may find it helpful see which API calls Tower makes in sequence. To do this, you can use the UI from Firebug or Chrome with developer plugins.

Another alternative is Charles Proxy (http://www.charlesproxy.com/), which offers a visualizer that you may find helpful. While it is commercial software, it can insert itself as an OS X proxy, for example, and intercept both requests from web browsers as well as curl and other API consumers.

Other alternatives include:

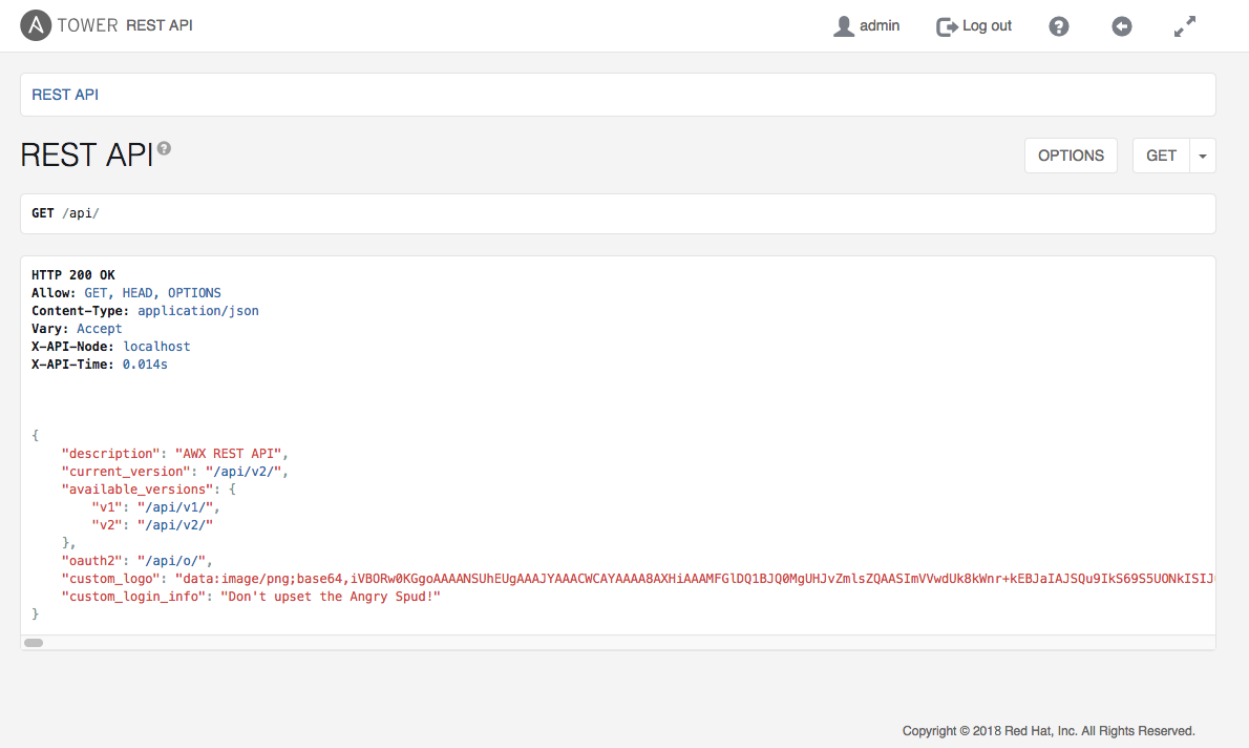
- Fiddler (http://www.telerik.com/fiddler)
- mitmproxy (https://mitmproxy.org/)
- Live HTTP headers FireFox extension (https://addons.mozilla.org/en-US/firefox/addon/live-http-headers/)
- Paros (http://sourceforge.net/projects/paros/)

CHAPTER

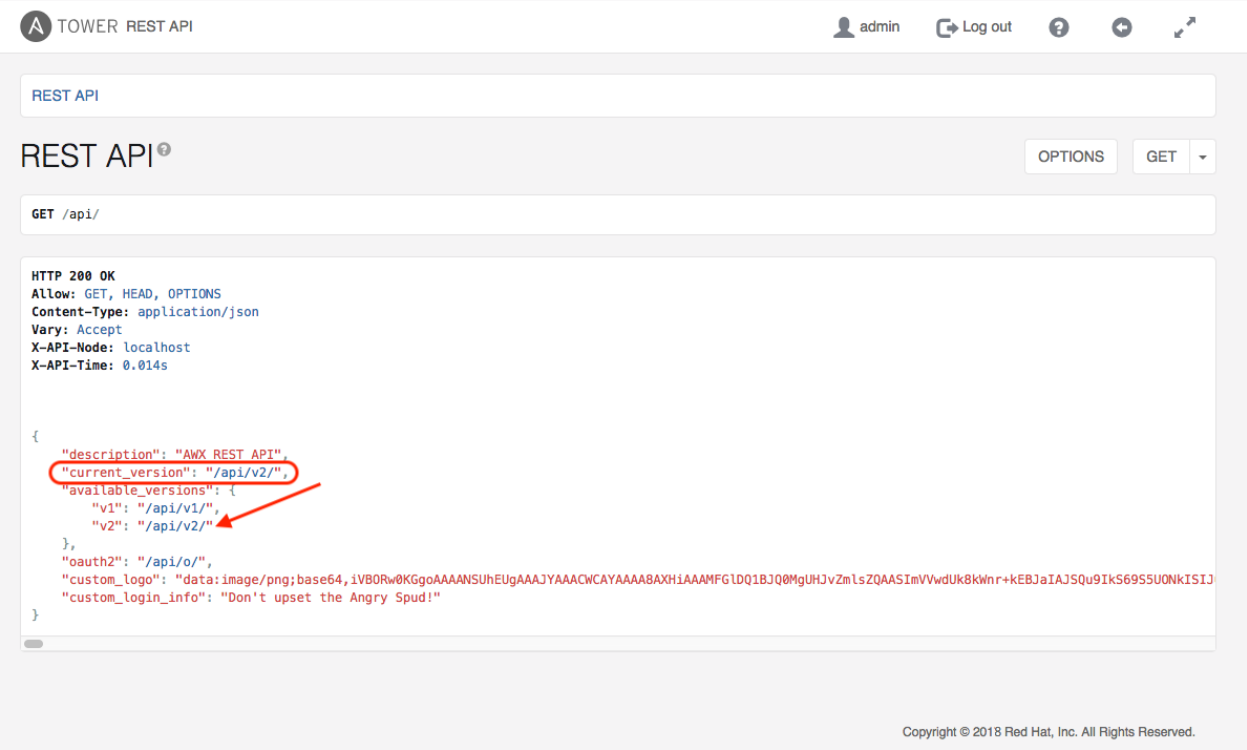
TWO

BROWSABLE API

REST APIs provide access to resources (data entities) via URI paths. You can visit the Ansible Tower REST API in a web browser at: `http://<Tower server name>/api/`



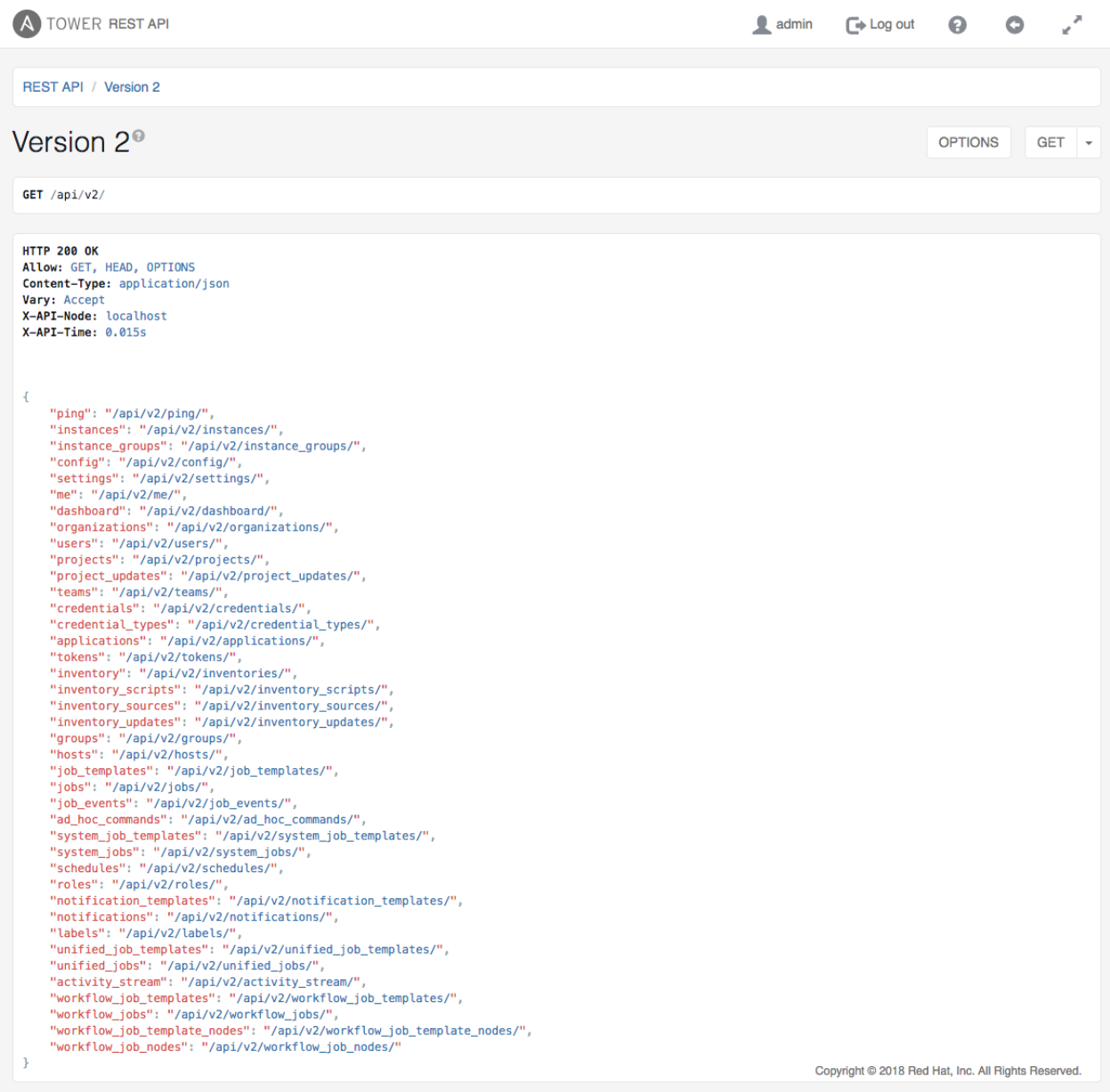
Ansible Tower 3.2 introduced version 2 of the API, which can be accessed by clicking the v2 link next to "available versions":



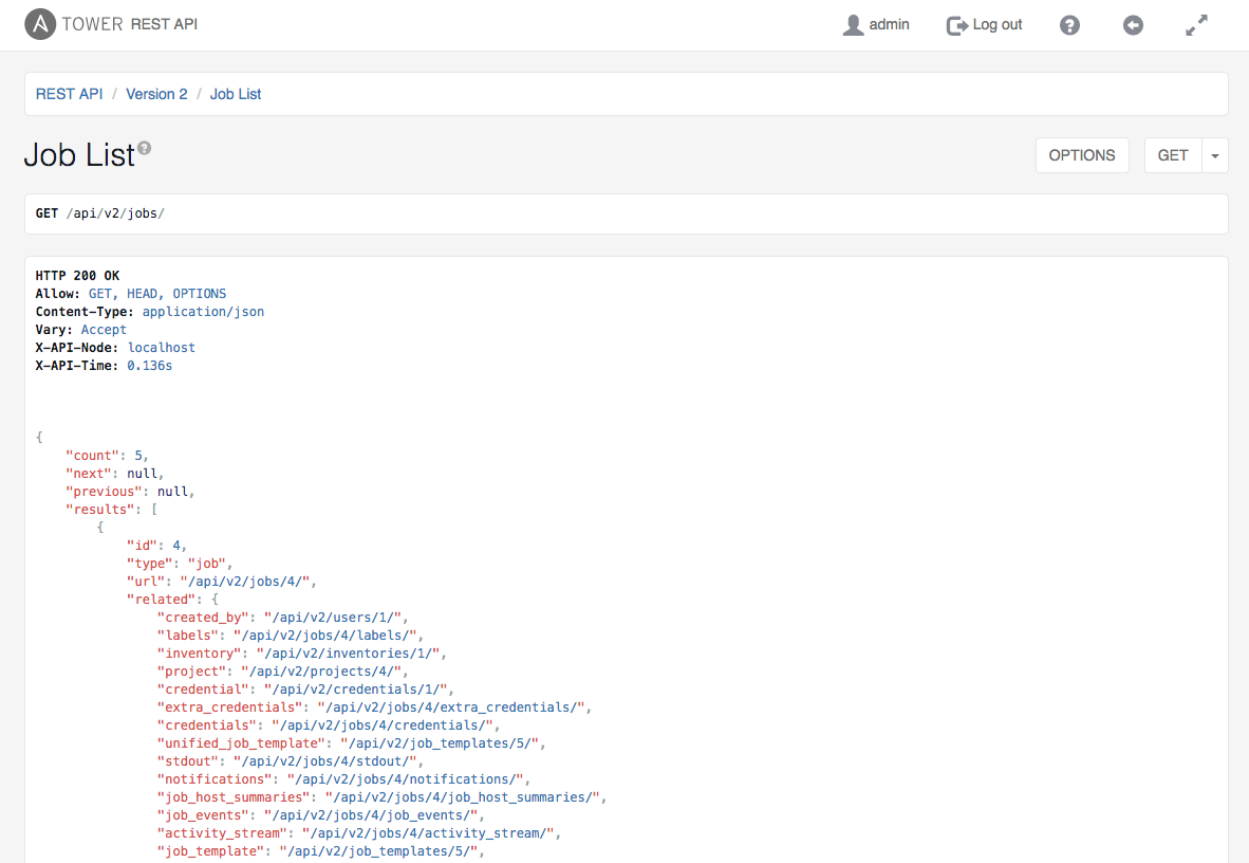
Alternatively, you can still access version 1 of the API this way.

If you perform a **GET** just the `/api/` endpoint, it gives the `current_version`, which would be the recommended version.

Clicking on various links in the API allows you to explore related resources.

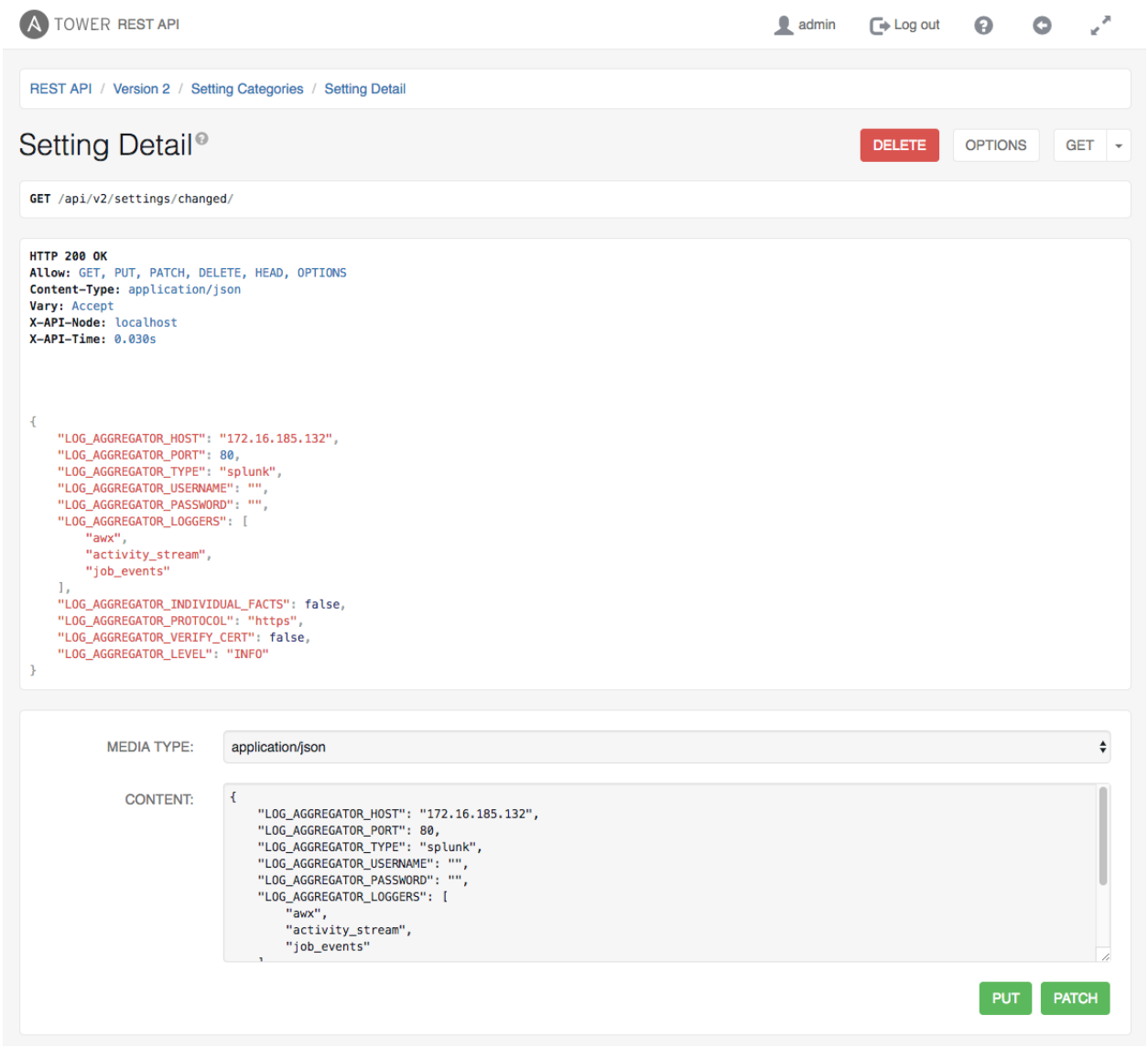


Clicking on the  next to the page name (toward the top of the screen) for an API endpoint gives you documentation on the access methods for that particular API endpoint and what data is returned when using those methods.



You can also use PUT and POST verbs on the specific API pages by formatting JSON in the various text fields.

You can also view changed settings from factory defaults at `/api/v2/settings/changed/` endpoint. It reflects changes you made in the API browser, not changed settings that come from static settings files.


TOWER REST API
admin
Log out
REST API / Version 2 / Setting Categories / Setting Detail
Setting Detail
DELETE
OPTIONS
GET
GET /api/v2/settings/changed/
HTTP 200 OK
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
X-API-Node: localhost
X-API-Time: 0.030s
{
"LOG_AGGREGATOR_HOST": "172.16.185.132",
"LOG_AGGREGATOR_PORT": 80,
"LOG_AGGREGATOR_TYPE": "splunk",
"LOG_AGGREGATOR_USERNAME": "",
"LOG_AGGREGATOR_PASSWORD": "",
"LOG_AGGREGATOR_LOGGERS": [
"awx",
"activity_stream",
"job_events"
],
"LOG_AGGREGATOR_INDIVIDUAL_FACTS": false,
"LOG_AGGREGATOR_PROTOCOL": "https",
"LOG_AGGREGATOR_VERIFY_CERT": false,
"LOG_AGGREGATOR_LEVEL": "INFO"
}
MEDIA TYPE:
application/json
CONTENT:
{
"LOG_AGGREGATOR_HOST": "172.16.185.132",
"LOG_AGGREGATOR_PORT": 80,
"LOG_AGGREGATOR_TYPE": "splunk",
"LOG_AGGREGATOR_USERNAME": "",
"LOG_AGGREGATOR_PASSWORD": "",
"LOG_AGGREGATOR_LOGGERS": [
"awx",
"activity_stream",
"job_events"
PUT
PATCH

CHAPTER

THREE

CONVENTIONS

Tower uses a standard REST API, rooted at `/api/` on the server. The API is versioned for compatibility reasons, `api/v2/` is currently available, and `/api/v1/` is deprecated and will be removed in a later release. You can see information about what API versions are available by querying `/api/`.

You may have to specify the content/type on **POST** or **PUT** requests accordingly.

- **PUT**: Update a specific resource (by an identifier) or a collection of resources. PUT can also be used to create a specific resource if the resource identifier is know before-hand.
- **POST**: Create a new resource. Also acts as a catch-all verb for operations that do not fit into the other categories.

All URIs not ending with `"/"` receive a 301 redirect.

Note: Ansible Tower 3.3.4 API change: Formatting of extra_vars attached to Job Template records is preserved. Previously, YAML would be converted to JSON and returned as JSON. In 2.2.0 and newer, YAML is returned as YAML with formatting and comments preserved, and JSON is returned as JSON.

CHAPTER

FOUR

SORTING

To provide examples that are easy to follow, the following URL is used throughout this guide:

```
http://<Tower server name>/api/v2/groups/
```

To specify that {{ model_verbose_name_plural }} are returned in a particular order, use the `order_by` query string parameter on the **GET** request.

```
http://<Tower server name>/api/v2/model_verbose_name_plural?order_by={{ order_field }}
```

Prefix the field name with a dash (-) to sort in reverse:

```
http://<Tower server name>/api/v2/model_verbose_name_plural?order_by=-{{ order_field }
↪}
```

Multiple sorting fields may be specified by separating the field names with a comma (`,`):

```
http://<Tower server name>/api/v2/model_verbose_name_plural?order_by={{ order_field }}
↪,some_other_field
```

CHAPTER

FIVE

SEARCHING

Use the search query string parameter to perform a case-insensitive search within all designated text fields of a model (added in AWX 1.4):

```
http://<Tower server name>/api/v2/model_verbose_name?search=findme
```

Search across related fields (added in AWX 1.4 / Ansible Tower 3.1):

```
http://<Tower server name>/api/v2/model_verbose_name?related__search=findme
```

CHAPTER

SIX

FILTERING

Any collection is what the system calls a "queryset" and can be filtered via various operators.

For example, to find the groups that contain the name "foo":

```
http://<Tower server name>/api/v2/groups/?name__contains=foo
```

To find an exact match:

```
http://<Tower server name>/api/v2/groups/?name=foo
```

If a resource is of an integer type, you must add `\_\_int` to the end to cast your string input value to an integer, like so:

```
http://<Tower server name>/api/v2/arbitrary_resource/?x__int=5
```

Related resources can also be queried, like so:

```
http://<Tower server name>/api/v2/users/?first_name__icontains=kim
```

This will return all users with names that include the string "Kim" in them.

You can also filter against multiple fields at once:

```
http://<Tower server name>/api/v2/groups/?name__icontains=test&has_active_
↪failures=false
```

This finds all groups containing the name "test" that has no active failures.

For more about what types of operators are available, refer to: https://docs.djangoproject.com/en/dev/ref/models/querysets/

Note: You can also watch the API as the UI is being used to see how it is filtering on various criteria.

Any additional query string parameters may be used to filter the list of results returned to those matching a given value. Only fields and relations that exist in the database may be used for filtering. Any special characters in the specified value should be url-encoded. For example:

```
?field=value%20xyz
```

Fields may also span relations, only for fields and relationships defined in the database:

```
?other__field=value
```

To exclude results matching certain criteria, prefix the field parameter with `not__`:

```
?not__field=value
```

(Added in AWX 1.4) By default, all query string filters are AND'ed together, so only the results matching all filters will be returned. To combine results matching any one of multiple criteria, prefix each query string parameter with `or__`:

```
?or__field=value&or__field=othervalue
?or__not__field=value&or__field=othervalue
```

(Added in Ansible Tower 1.4.5) The default AND filtering applies all filters simultaneously to each related object being filtered across database relationships. The chain filter instead applies filters separately for each related object. To use, prefix the query string parameter with `chain__`:

```
?chain__related__field=value&chain__related__field2=othervalue
?chain__not__related__field=value&chain__related__field2=othervalue
```

If the first query above were written as `?related__field=value&related__field2=othervalue`, it would return only the primary objects where the same related object satisfied both conditions. As written using the chain filter, it would return the intersection of primary objects matching each condition.

Field lookups may also be used for more advanced queries, by appending the lookup to the field name:

```
?field__lookup=value
```

The following field lookups are supported:

- `exact`: Exact match (default lookup if not specified).
- `iexact`: Case-insensitive version of exact.
- `contains`: Field contains value.
- `icontains`: Case-insensitive version of contains.
- `startswith`: Field starts with value.
- `istartswith`: Case-insensitive version of startswith.
- `endswith`: Field ends with value.
- `iendswith`: Case-insensitive version of endswith.
- `regex`: Field matches the given regular expression.
- `iregex`: Case-insensitive version of regex.
- `gt`: Greater than comparison.
- `gte`: Greater than or equal to comparison.
- `lt`: Less than comparison.
- `lte`: Less than or equal to comparison.
- `isnull`: Check whether the given field or related object is null; expects a boolean value.
- `in`: Check whether the given field's value is present in the list provided; expects a list of items.
- Boolean values may be specified as `True` or `1` for true, `False` or `0` for false (both case-insensitive).

Null values may be specified as `None` or `Null` (both case-insensitive), though it is preferred to use the `isnull` lookup to explicitly check for null values.

Lists (for the `in` lookup) may be specified as a comma-separated list of values.

Filtering based on the requesting user's level of access by query string parameter (added in Ansible Tower 3.1):

- `role_level`: Level of role to filter on, such as `admin_role`

CHAPTER

SEVEN

PAGINATION

Responses for collections in the API are paginated. This means that while a collection may contain tens or hundreds of thousands of objects, in each web request, only a limited number of results are returned for API performance reasons.

When you get back the result for a collection you will see something similar to the following:

```
{'count': 25, 'next': 'http://testserver/api/v2/some_resource?page=2', 'previous':␣
↪None, 'results': [ ... ] }
```

To get the next page, simply request the page given by the 'next' sequential URL.

Use the `page_size=XX` query string parameter to change the number of results returned for each request.

Use the `page` query string parameter to retrieve a particular page of results.

```
http://<Tower server name>/api/v2/model_verbose_name?page_size=100&page=2
```

The previous and next links returned with the results will set these query string parameters automatically.

The serializer is quite efficient, but you should probably not request page sizes beyond a couple of hundred.

The user interface uses smaller values to avoid the user having to do a lot of scrolling.

CHAPTER

EIGHT

ACCESS RESOURCES

Traditionally, Ansible Tower uses a primary key to access individual resource objects. Starting in 3.2 and API v2, the named URL feature allows you to access Tower resources via resource-specific human-readable identifiers. In Ansible Tower versions prior to 3.2, the only way of accessing a resource object without auxiliary query string is via resource primary key number, for example, via URL path: `/api/v2/hosts/2/`. Now, you can use a named URL to do the same thing, for example, via URL path `/api/v2/hosts/host_name++inv_name++org_name/`.

8.1 Configuration Settings

There are two named-URL-related Tower configuration settings available under `/api/v2/settings/named-url/`:

`NAMED_URL_FORMATS` and `NAMED_URL_GRAPH_NODES`

`NAMED_URL_FORMATS` is a read only key-value pair list of all available named URL identifier formats. A typical `NAMED_URL_FORMATS` looks like this:

```
"NAMED_URL_FORMATS": {
"job_templates": "<name>",
"workflow_job_templates": "<name>",
"inventories": "<name>++<organization.name>",
"users": "<username>",
"custom_inventory_scripts": "<name>++<organization.name>",
"labels": "<name>++<organization.name>",
"credential_types": "<name>+<kind>",
"notification_templates": "<name>++<organization.name>",
"instances": "<hostname>",
"instance_groups": "<name>",
"hosts": "<name>++<inventory.name>++<organization.name>",
"system_job_templates": "<name>",
"groups": "<name>++<inventory.name>++<organization.name>",
"organizations": "<name>",
"credentials": "<name>++<credential_type.name>+<credential_type.kind>++<organization.
↪name>",
"teams": "<name>++<organization.name>",
"inventory_sources": "<name>",
"projects": "<name>"
}
```

For each item in `NAMED_URL_FORMATS`, the key is the API name of the resource to have named URL, the value is a string indicating how to form a human-readable unique identifier for that resource. `NAMED_URL_FORMATS` exclusively lists every resource that can have named URL, any resource not listed there has no named URL. If a resource can have named URL, its objects should have a named_url field which represents the object-specific named URL. That

field should only be visible under detail view, not list view. You can access specified resource objects using accurately generated named URL. This includes not only the object itself but also its related URLs. For example, if `/api/v2/res_name/obj_slug/` is valid, `/api/v2/res_name/obj_slug/related_res_name/` should also be valid.

`NAMED_URL_FORMATS` are instructive enough to compose human-readable unique identifier and named URL themselves. For ease-of-use, every object of a resource that can have named URL will have a related field `named_url` that displays that object's named URL. You can copy and paste that field for your own custom use. Also refer to the help text of API browser if a resource object has named URL for further guidance.

Suppose you want to manually determine the named URL for a label with ID 5. A typical procedure of composing a named URL for this specific resource object using `NAMED_URL_FORMATS` is to first look up the labels field of `NAMED_URL_FORMATS` to get the identifier format `<name>++<organization.name>`:

- The first part of the URL format is `<name>`, which indicates that the label resource detail can be found in `/api/v2/labels/5/`, and look for `name` field in returned JSON. Suppose you have the `name` field with value 'Foo', then the first part of the unique identifier is **Foo**.
- The second part of the format are double pluses ++. That is the delimiter that separates different parts of a unique identifier. Append them to the unique identifier to get **Foo++**
- The third part of the format is `<organization.name>`, which indicates that field is not in the current label object under investigation, but in an organization which the label object points to. Thus, as the format indicates, look up the organization in the related field of current returned JSON. That field may or may not exist. If it exists, follow the URL given in that field, for example, `/api/v2/organizations/3/`, to get the detail of the specific organization, extract its `name` field, for example, 'Default', and append it to our current unique identifier. Since `<organizations.name>` is the last part of format, thus, generating the resulting named URL: `/api/v2/labels/Foo++Default/`. In the case where organization does not exist in related field of the label object detail, append an empty string instead, which essentially does not alter the current identifier. So `Foo++` becomes the final unique identifier and the resulting generated named URL becomes `/api/v2/labels/Foo++/`.

An important aspect of generating a unique identifier for named URL has to do with reserved characters. Because the identifier is part of a URL, the following reserved characters by URL standard is encoded by percentage symbols: `;/?:@=&[]`. For example, if an organization is named `;/?:@=&[]`, its unique identifier should be `%3B%2F%3F%3A%40%3D%26%5B%5D`. Another special reserved character is +, which is not reserved by URL standard but used by named URL to link different parts of an identifier. It is encoded by `[+]`. For example, if an organization is named `[+]`, its unique identifier is `%5B[+]%5D`, where original `[` and `]` are percent encoded and + is converted to `[+]`.

Although `NAMED_URL_FORMATS` cannot be manually modified, modifications do occur automatically and expanded over time, reflecting underlying resource modification and expansion. Consult the `NAMED_URL_FORMATS` on the same Tower cluster where you want to use the named URL feature.

`NAMED_URL_GRAPH_NODES` is another read-only list of key-value pairs that exposes the internal graph data structure Tower used to manage named URLs. This is not intended to be human-readable but should be used for programmatically generating named URLs. An example script for generating named URL given the primary key of arbitrary resource objects that can have a named URL, using info provided by `NAMED_URL_GRAPH_NODES`, can be found in GitHub at https://github.com/ansible/awx/blob/devel/tools/scripts/pk_to_named_url.py.

8.2 Identifier Format Protocol

Resources in Tower are identifiable by their unique keys, which are basically tuples of resource fields. Every Tower resource is guaranteed to have its primary key number alone as a unique key, but there might be multiple other unique keys. A resource can generate an identifier format thus, have a named URL if it contains at least one unique key that satisfies the rules below:

1. The key must contain only fields that are either the `name` field, or text fields with a finite number of possible choices (like credential type resource's `kind` field).
2. The only allowed exceptional fields that breaks rule #1 is a many-to-one related field relating to a resource other than itself, which is also allowed to have a slug.

Suppose Tower has resources `Foo` and `Bar`, both `Foo` and `Bar` contain a `name` field and a `choice` field that can only have value 'yes' or 'no'. Additionally, resource `Foo` contains a many-to-one field (a foreign key) relating to `Bar`, e.g. `fk`. `Foo` has a unique key tuple (`name`, `choice`, `fk`) and `Bar` has a unique key tuple (`name`, `choice`). `Bar` can have named URL because it satisfies rule #1 above. `Foo` can also have named URL, even though it breaks rule #1, the extra field breaking rule #1 is the `fk` field, which is many-to-one-related to `Bar` and `Bar` can have named URL.

For resources satisfying rule #1 above, their human-readable unique identifiers are combinations of foreign key fields, delimited by +. In specific, resource `Bar` in the above example will have slug format `<name>+<choice>`. Note the field order matters in slug format: `name` field always comes first if present, following by all the rest fields arranged in lexicographic order of field name. For example, if Bar also has an `a_choice` field satisfying rule #1 and the unique key becomes (`name`, `choice`, `a_choice`), its slug format becomes `<name>+<a_choice>+<choice>`.

For resources satisfying rule #2 above, if traced back via the extra foreign key fields, the result is a tree of resources that all together identify objects of that resource. In order to generate identifier format, each resource in the traceback tree generates its own part of standalone format in the way previously described, using all fields but the foreign keys. Finally all parts are combined by ++ in the following order:

- Put stand-alone format as the first identifier component.
- Recursively generate unique identifiers for each resource. The underlying resource is pointing to using a foreign key (a child of a traceback tree node).
- Treat generated unique identifiers as the rest of the identifier components. Sort them in lexicographic order of corresponding foreign keys.
- Combine all components together using ++ to generate the final identifier format.

In reference to the example above, when generating an identifier format for resource `Foo`, Tower generates the stand-alone formats, `<name>+<choice>` for `Foo` and `<fk.name>+<fk.choice>` for `Bar`, then combine them together to be `<name>+<choice>++<fk.name>+<fk.choice>`.

When generating identifiers according to the given identifier format, there are cases where a foreign key may point to nowhere. In this case, Tower substitutes the part of the format corresponding to the resource the foreign key should point to with an empty string ''. For example, if a `Foo` object has the name ='alice', choice ='yes', but `fk` field = None, its resulting identifier will be `alice+yes++`.

CHAPTER

NINE

READ-ONLY FIELDS

Certain fields in the REST API are marked read-only. These usually include the URL of a resource, the ID, and occasionally some internal fields. For instance, the `'created\_by'` attribute of each object indicates which user created the resource, and cannot be edited.

If you post some values and notice that they are not changing, these fields may be read-only.

CHAPTER

TEN

INTRODUCTION TO TOWER-CLI

tower-cli is a command line tool for Ansible Tower. It allows Tower commands to be easily run from the UNIX command line. It can also be used as a client library for other python apps, or as a reference for others developing API interactions with Tower's REST API.

Note: The `tower-cli` software is an open source project currently under development and, until a complete implementation occurs, only implements a subset of Tower's features.

10.1 License

While Tower is commercially licensed software, tower-cli is an open source project. Specifically, this project is licensed under the Apache 2.0 license. Pull requests, contributions, and tickets filed in GitHub are warmly welcomed.

10.2 Capabilities

`tower-cli` sends commands to the Tower API. It is capable of retrieving, creating, modifying, and deleting most objects within Tower.

A few potential uses include:

- Launching playbook runs (for instance, from Jenkins, TeamCity, Bamboo, etc)
- Checking on job statuses
- Rapidly creating objects like organizations, users, teams, and more

10.3 Installation

`tower-cli` is available as a package on PyPI.

The preferred way to install is through `pip`:

```
$ pip install ansible-tower-cli
```

The main branch of this project may also be consumed directly from source.

For more information on `tower-cli`, refer to the project page at: https://github.com/ansible/tower-cli/

10.4 Configuration

`tower-cli` can edit its own configuration or users can directly edit the configuration file, allowing configuration to be set in multiple ways.

10.4.1 Set configuration with tower-cli config

The preferred way to set configuration is with the `tower-cli config` command.

```
$ tower-cli config key value
```

By issuing the `tower-cli config` command without arguments, you can view a full list of configuration options and where they are set. The default behavior allows environment variables to override your tower-cli.cfg settings, but they will not override configuration values that are passed in on the command line at runtime. The available environment variables and their corresponding Tower configuration keys are as follows:

- TOWER_COLOR: color
- TOWER_FORMAT: format
- TOWER_HOST: host
- TOWER_PASSWORD: password
- TOWER_USERNAME: username
- TOWER_VERIFY_SSL: verify_ssl
- TOWER_VERBOSE: verbose
- TOWER_DESCRIPTION_ON: description_on
- TOWER_CERTIFICATE: certificate

You will generally need to set at least three configuration options–host, username, and password–which correspond to the location of your Ansible Tower instance and your credentials to authenticate to Tower.

```
$ tower-cli config host tower.example.com
$ tower-cli config username leeroyjenkins
$ tower-cli config password myPassw0rd
```

10.4.2 Write to the config files directly

The configuration file can also be edited directly. A configuration file is a simple file with keys and values, separated by `:` or `=`:

```
host: tower.example.com
username: admin
password: p4ssw0rd
```

10.4.3 File Locations

The order of precedence for configuration file locations is as follows, from least to greatest:

- internal defaults
- `/etc/tower/tower_cli.cfg` (written using `tower-cli config --global`)

- `~/.tower_cli.cfg` (written using `tower-cli config`)
- run-time parameters

10.4.4 Usage

`tower-cli` invocation generally follows this format:

```
$ tower-cli {resource} {action} ...
```

The **resource** is a type of object within Tower (a noun), such as `user`, `organization`, `job_template`, etc.; resource names are always singular in Tower CLI (use `tower-cli user`, never `tower-cli users`).

The **action** is the thing you want to do (a verb). Most `tower-cli` resources have the following actions–`get`, `list`, `create`, `modify`, and `delete`–and have options corresponding to fields on the object in Tower.

Examples of actions and resources include (but are not limited to):

User Actions

```
# List all users.
$ tower-cli user list

# List all non-superusers
$ tower-cli user list --is-superuser=false

# Get the user with the ID of 42.
$ tower-cli user get 42

# Get the user with the given username.
$ tower-cli user get --username=guido

# Create a new user.
$ tower-cli user create --username=guido --first-name=Guido \
                        --last-name="Van Rossum" --email=guido@python.org

# Modify an existing user.
# This would modify the first name of the user with the ID of "42" to "Guido".
$ tower-cli user modify 42 --first-name=Guido

# Modify an existing user, lookup by username.
# This would use "username" as the lookup, and modify the first name.
# Which fields are used as lookups vary by resource, but are generally
# the resource's name.
$ tower-cli user modify --username=guido --first-name=Guido

# Delete a user.
$ tower-cli user delete 42
```

Job Actions

```
# Launch a job.
$ tower-cli job launch --job-template=144
```

```
# Monitor a job.
$ tower-cli job monitor 95
```

Group Actions

```
# Get a list of groups.
$ tower-cli group list

# Sync a group by the groupID.
$ tower-cli group sync $groupID
```

When in doubt, check the help for more options:

```
$ tower-cli # help
$ tower-cli user --help # resource specific help
$ tower-cli user create --help # command specific help
```

Workflow Actions

Starting with Tower 3.1.0 and Tower-CLI 3.1.0, workflow networks can be managed from Tower-CLI either by normal CRUD actions or by using a YAML file that defines the workflow network.

```
# Print out the schema for a workflow
$ tower-cli workflow schema workflow_name

# Create the network defined in file "schema.yml"
$ tower-cli workflow schema workflow_name @schema.yml
```

The following is an example of what a schema might look like.

```
- job_template: Hello world
  failure:
  - inventory_source: AWS servers (AWS servers - 42)
  success:
  - project: Ansible Examples
    always:
    - job_template: Echo variable
      success:
      - job_template: Scan localhost
```

For more details, see the tower-cli workflow doc at https://github.com/ansible/tower-cli/blob/master/docs/WORKFLOWS.md

10.4.5 Specify extra variables

There are a number of ways to pass extra variables to the Tower server when launching a job:

- Pass data in a file using the flag `--extra-vars="@filename.yml"`
- Include yaml data at runtime with the flag `--extra-vars="var:  value"`
- A command line editor automatically pops up when the job template is marked to prompt on launch
- If the job template has extra variables, these are not over-ridden

These methods can also be combined. For instance, if you give the flag multiple times on the command line, specifying a file in addition to manually giving extra variables, these two sources are combined and sent to the Tower server.

```
# Launch a job with extra variables from filename.yml, and also a=5
$ tower-cli job launch --job-template=1 --extra-vars="a=5 b=3" \
                                        --extra-vars="@filename.yml"

# Create a job template with that same set of extra variables
$ tower-cli job_template create --name=test_job_template --project=1 \
                                --inventory=1 --playbook=helloworld.yml \
                                --machine-credential=1 --extra-vars="a=5 b=3" \
                                --extra-vars="@filename.yml"
```

You may not combine multiple sources when modifying a job template. Whitespace can be used in strings like `--extra-vars="a='white space'"`, and list-valued parameters can be sent as JSON or YAML, but not `key=value` pairs. For instance, `--extra-vars="a: [1, 2, 3, 4, 5]"` sends the parameter `"a"` with that list as its value.

Note: Additional strict `extra_vars` validation was added in Ansible Tower 3.0.0. `extra_vars` passed to the job launch API are only honored if one of the following is true:

- They correspond to variables in an enabled survey
- `ask_variables_on_launch` is set to True

10.4.6 SSL warnings

By default, tower-cli raises an error if the SSL certificate of the Tower server cannot be verified. To allow unverified SSL connections, set the config variable, `verify_ssl = false`. To allow it for a single command to override `verify_ssl` if set to `true`, add the `--insecure` flag:

```
# Disable insecure connection warnings permanently
$ tower-cli config verify_ssl false

# Disable insecure connection warnings for just this command
$ tower-cli job_template list --insecure
```

CHAPTER

ELEVEN

TOWER API REFERENCE GUIDE

The Ansible Tower API Reference Manual provides in-depth documentation for Tower's REST API, including examples on how to integrate with it.

CHAPTER

TWELVE

INDEX

- genindex

CHAPTER

THIRTEEN

COPYRIGHT © 2019 RED HAT, INC.

INDEX

A

access resources, 15
API
    browsable, 3
    JSON, 6
    POST, 6
    PUT, 6
    root directory, 8

B

browsable API, 3

C

content type
    JSON, 8
conventions, 8

F

filtering, 11

J

JSON
    API, 6
    content type, 8

O

ordering
    sorting, 9

P

pagination, 14
POST
    API, 6
PUT
    API, 6

Q

queryset, 11

R

read-only fields, 18
root directory
    API, 8

S

searching, 10
serializer, 14
sorting
    ordering, 9

T

tools, 2
    tower-cli, 19
tower-cli, 19
    capabilities, 19
    installation, 19