ZTPServer Documentation

Release 1.1.0

Arista Networks

February 09, 2015

Contents

ZTPServer provides a bootstrap environment for Arista EOS based products. It is written mostly in Python and leverages standard protocols like DHCP (for boot functions), HTTP (for bi-directional transport), XMPP and syslog (for logging). Most of the configuration files are YAML based.

This open source project is maintained by the Arista Networks EOS+ services organization.

CHAPTER 1

Highlights

- Extends the basic capability of EOS's zero-touch provisioning feature in order to allow more robust provisioning activities
- Is extensible, for easy integration into various network environments
- Can be run natively in EOS or any Linux server
- Arista EOS+ led community open source project

CHAPTER 2

Features

- Dynamic startup-config generation and automatic install
- Image and file system validation and standardization
- Connectivity validation and topology based auto-provisioning
- Config and device templates with dynamic resource allocation
- Zero-touch replacement and upgrade capabilities
- User extensible actions
- Email, XMPP, syslog based

2.1 Overview

ZTPServer provides a robust server which enables comprehensive bootstrap solutions for Arista EOS based network elements. ZTPserver interacts with the ZeroTouch Provisioning (ZTP) mode of EOS which takes an unprovisioned network element to a bootstrap ready state whenever a valid configuration file is not present on the internal flash storage.

ZTPServer provides a number of features that extend beyond simply loading a configuration file and and boot image on a node, including:

- sending an advanced bootstrap client to the node: the bootstrap script.
- mapping each node to an individual definition which describes the bootstrap steps specific to that node
- defining configuration templates and actions which can be shared by multiple nodes - the actions can be customised using statically defined or dynamically generated attributes
- implementing environment-specific actions which integrate with external systems
- topology validation using a simple syntax to express LLDP neighbor adjacencies

ZTPServer is written in Python and leverages standard protocols like DHCP (DHCP options for boot functions), HTTP(S) (for bi-directional transport), XMPP and syslog (for logging). Most of the configuration files are YAML based.

Highlights:

- Extends the basic capability of EOS ZTP to allow more robust provisioning activities
- Extensible for easy integration into network operational environment
- Can be run natively in EOS or on a separate server.

- An Arista EOS+ led community open source project

Features:

- Automated configuration file generation and application
- Image and file system validation and standardization
- Connectivity validation and topology based auto-provisioning
- Configuration and device templates with resource allocation for dynamic deployments
- Zero Touch Replacement and upgrade capabilities
- User extensible actions
- Email, XMPP, syslog based logging and accounting of all processes

2.1.1 ZTP Intro

Zero Touch Provisioning (ZTP) is a feature in Arista EOS software which, in the absence of a startup-config, attempts to configure a switch over the network.

The basic flow is as follows:

- Check for startup-config, if absent, attempt ZTP
- Send out a DHCP request on all connected interfaces
- If a DHCP response is received with Option 67 defined (bootfile-name), retrieve that file
- If that file is a startup-config, then apply it to the device and boot
- If that file is an executable, then run it. Common actions include upgrading the EOS image, downloading extension packages, and dynamically building a startup-config file. (**ZTPServer's bootstrap script is launched this way**)
- Restart with the new configuration.

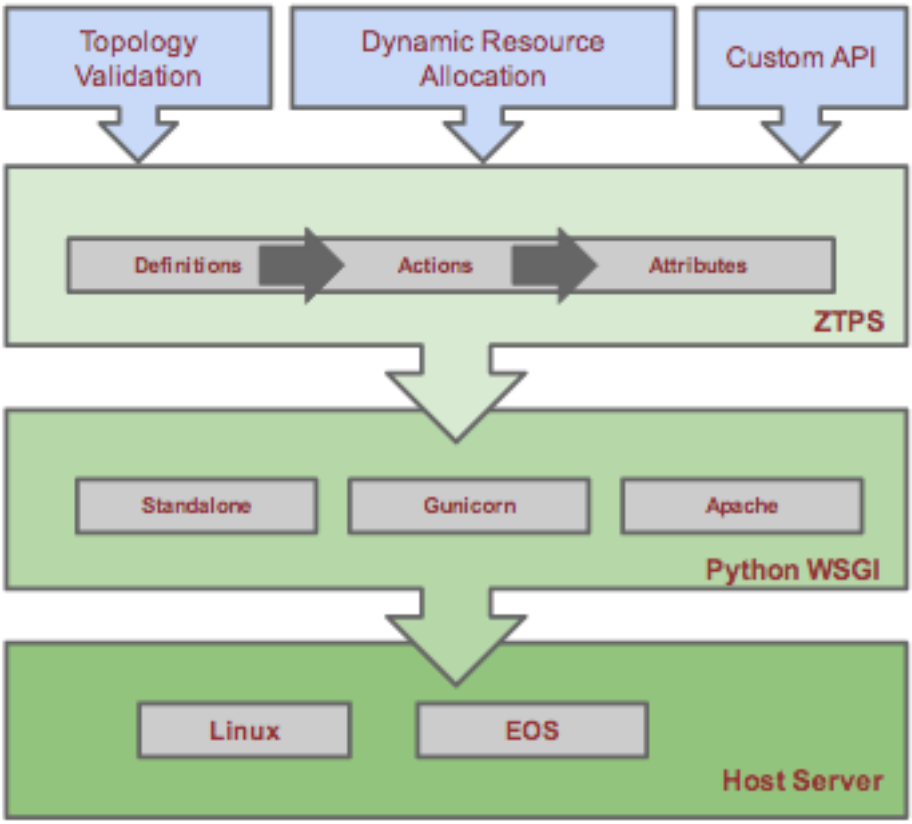
See the ZTP Tech Bulletin and the Press Release for more information on ZTP.

2.1.2 Architecture

There are 2 primary components of the ZTPServer implementation:

- the **server** or ZTPServer instance **AND**
- the **client** or bootstrap (a process running on each node, which connects to the server in order to provision the node)

2.1.3 Server



The server can run on any standard x86 server. Currently the only OS-es tested are Linux and MacOS, but theoretically any system that supports Python could run ZTPServer. The server provides a Python WSGI compliant interface, along with a standalone HTTP server. The built in HTTP server runs on port 8080 by default and provides bidirectional file transport for the bootstrap process.

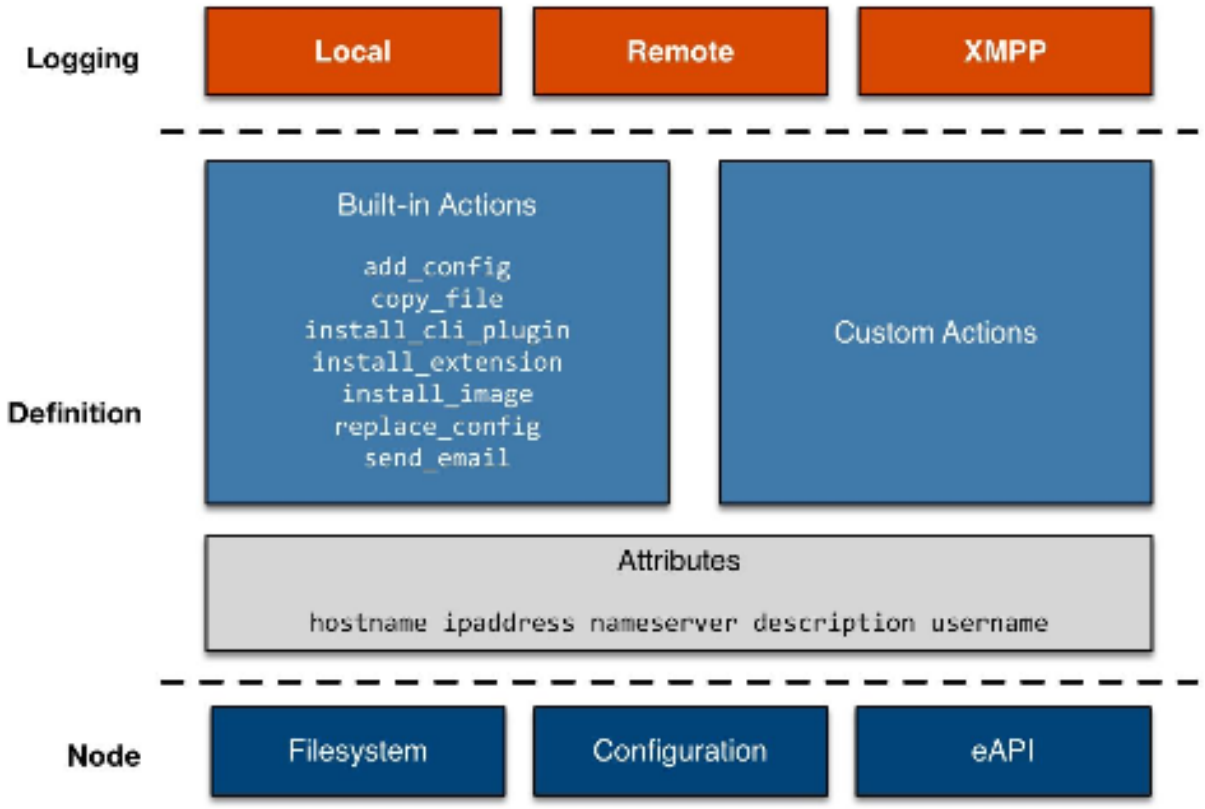
The primary methods of provisioning a node are:

- **statically** via predefined node entries OR
- **dynamically** generated via definitions and resource pools

Both methods can leverage topology validation via neighbordb and/or pattern entries.

The definition associated with each node contains a set of actions that can perform a variety of functions that ultimately lead to a final device configuration and file structure. Actions can use statically configured attributes or leverage configuration templates and dynamically allocated attributes to generate the system configuration. Definitions, actions, attributes, templates, and resources are all defined in YAML files.

2.1.4 Client



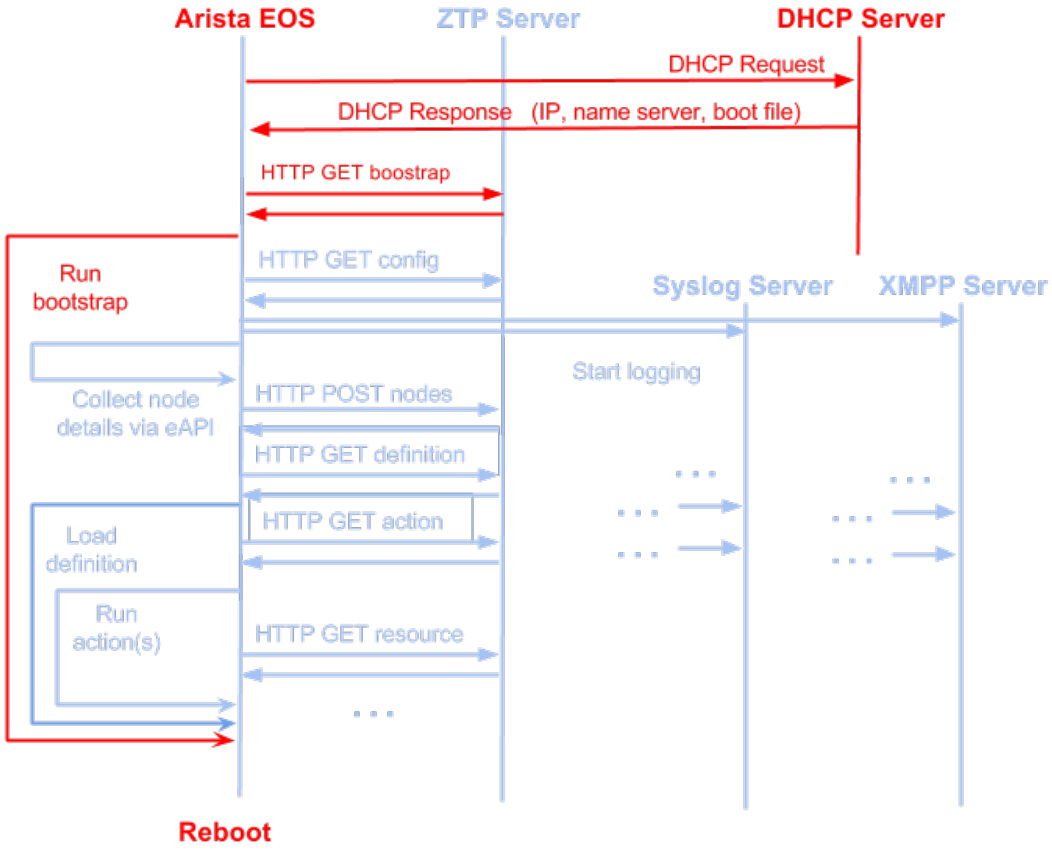
The client or **bootstrap file** is retrieved by the node via an HTTP GET request made to the ZTPServer (the URL of the file is retrieved via DHCP option 67). This file executes locally and gathers system and LLDP neighbor information from the unprovisioned device and returns it to the ZTPServer. Once the ZTPServer processes the information and confirms that it can provision the node, the client makes a request to the server for a definition file - this file will contain the list of all actions which need to be executed by the node in order to provision itself.

Throughout the provisioning process the bootstrap client can log via both local and remote logging and XMPP.

2.1.5 ZTP Client-Server Message Flows

A high level view of the client - server message flows can be seen in the following diagram:

(Red indicates Arista EOS flows. Blue indicates the bootstrap client.)


Arista EOS
ZTP Server
DHCP Server
DHCP Request
DHCP Response (IP, name server, boot file)
HTTP GET boostrap
Run bootstrap
HTTP GET config
Syslog Server
XMPP Server
Start logging
Collect node details via eAPI
HTTP POST nodes
HTTP GET definition
HTTP GET action
Load definition
Run action(s)
HTTP GET resource
Reboot

2.1.6 Topology Validation

```
- name: standard leaf definition
  definition: dc-1/pod-1/leaf_template
  variables:
    - not_spine: excludes('spine')
    - any_spine: regex('spine\d+')
    - any_pod: includes('pod')
  interfaces:
    - Ethernet1: any_spine:Ethernet1/1
    - Ethernet2: pod1-spine2:any
    - any: excludes('spine1'):Ethernet49
    - any: excludes('spine2'):Ethernet49
    - Ethernet49:
        device: not_spine
        port: en0
    - Ethernet50:
        device: includes('spine')
         port: Ethernet50
```

ZTPServer provides a powerful topology validation engine via `neighbordb` or pattern files. As part of the bootstrap process for each node, the LLDP information received on all ports is passed to the ZTPServer and pattern matched against either `neighbordb` or a node-specific pattern file (if a node is already configured on the server). Both are YAML files that are use a simple format to express strongly and loosely typed topology patterns. Pattern entries are processed top down and can include local or globally-defined variables (including regular expressions).

Patterns in `neighbordb` match nodes to definitions (dynamic mode), while node-specific pattern files are used for cabling and connectivity validation (static mode).

Topology-validation can be disabled:

- globally (`disable_topology_validation=true` in the server's global configuration file) OR
- on a per-node basis, by adding a pattern which matches any topology

2.1.7 Operational modes

There are 4 operational modes for ZTPServer, explained below. See *Mode Examples* to see how to use them.

Statically defined node without topology validation

- Node is created in /nodes/<unique_id>/ before bootstrap
- Definition or startup-config is placed in /nodes

- Topology validation is disabled globally or with an open pattern

Statically defined node with topology validation

- Node is created in /nodes/<unique_id>/ before bootstrap
- Definition or startup-config is placed in /nodes
- Topology validation is enabled globally and pattern is placed in /nodes

Strongly-typed node with topology validation

- Definition is node specific, though the /nodes/<unique_id>/ directory is not pre-created
- /nodes/<unique_id>/ is dynamically created during ZTP provisioning
- Topology validation is enabled globally and pattern in neighbordb references the node's unique_id

Weakly-typed node with topology validation

- Definition is NOT node specific, leverages resources and templates
- /nodes/<unique_id>/ is dynamically created during ZTP provisioning
- Topology validation is enabled globally and pattern is matched in neighbordb

2.2 Installation

There are 3 primary installation methods:

- *Turn-key VM Creation*
- *PyPI Package (pip install)*
- *Manual installation*

Examples in this guide are based on the following:

- Python 2.7
- dhcp server - (dhcpd)
- pip
- git

2.2.1 Requirements

Server:

- Python 2.7 or later (https://www.python.org/download/releases)
- routes 2.0 or later (https://pypi.python.org/pypi/Routes)
- webob 1.3 or later (http://webob.org/)
- PyYaml 3.0 or later (http://pyyaml.org/)

Client:

- 4.12.0 or later (ZTPServer 1.1)
- 4.13.3 or later (ZTPServer 1.0)

Note: We recommend using a Linux distribution which has Python 2.7 as its standard Python install (e.g. yum in Centos requires Python 2.6 and a dual Python install can be fairly tricky and buggy). This guide was written based ZTPServer v1.1.0 installed on Fedora 20.

2.2.2 Turn-key VM Creation

The turn-key VM option leverages Packer to auto generate a VM on your local system. Packer.io automates the creation of the ZTPServer VM. All of the required packages and dependencies are installed and configured. The current Packer configuration allows you to choose between VirtualBox or VMWare as your hypervisor and each can support Fedora 20 or Ubuntu Server 12.04.

VM Specification:

- 7GB Hard Drive
- 2GB RAM
- Hostname ztps.ztps-test.com
  - eth0 (NAT) DHCP
  - eth1 (hostonly) 172.16.130.10
- Firewalld/UFW disabled.
- Users
  - root/eosplus
  - ztpsadmin/eosplus
- Python 2.7.5 with PIP
- DHCP installed with Option 67 configured (eth1 only)
- BIND DNS server installed with zone ztps-test.com
  - wildcard forwarding rule passing all other queries to 8.8.8.8
  - SRV RR for im.ztps-test.com
- rsyslog-ng installed; Listening on UDP and TCP (port 514)
- ejabberd (XMPP server) configured for im.ztps-test.com
  - XMPP admin user ztpsadmin, passwd eosplus

- httpd installed and configured for ZTPServer (mod_wsgi)
- ZTPServer installed
- ztpserver-demo repo files pre-loaded

See the Packer VM code and documentation as well as the ZTPServer demo files for the Packer VM.

2.2.3 PyPI Package (pip install)

ZTPServer may be installed as a PyPI package.

This option assumes you have a server with Python and pip pre-installed. See installing pip.

Once pip is installed, type:

```
bash-3.2$ pip install ztpserver
```

The pip install process will install all dependencies and run the install script, leaving you with a ZTPServer instance ready to configure.

2.2.4 Manual installation

Download:

Release	Git	ZIP	TAR
1.1.0 (Current)	GitHub	ZIP	TAR
Development (Unstable)	GitHub	ZIP	TAR

Once the above system requirements are met, use the following git command to pull the develop branch into a local directory on the server where you want to install ZTPServer:

```
bash-3.2$ git clone https://github.com/arista-eosplus/ztpserver.git
```

Or, you may download the zip or tar archive and expand it.

```
bash-3.2$ wget https://github.com/arista-eosplus/ztpserver/tarball/master
bash-3.2$ tar xvf <filename>
  or
bash-3.2$ unzip <filename>
```

Change in to the ztpserver directory, then checkout the release desired:

```
bash-3.2$ cd ztpserver
bash-3.2$ git checkout v1.1.0
```

Execute `setup.py` to build and then install ZTPServer

```
[user@localhost ztpserver]$ python setup.py build
running build
running build_py
...

[root@localhost ztpserver]# sudo python setup.py install
running install
running build
running build_py
running install_lib
...
```

2.2.5 Configure additional services

Configure the DHCP Service

Set up your DHCP infrastructure to server the full path to the ZTPServer bootstrap file via option 67. This can be performed on any DHCP server. Instructions are provided, below, for ISC dhcpd.

Get dhcpd:

RedHat: `bash-3.2$ sudo yum install dhcp`

Ubuntu: `bash-3.2$ sudo apt-get install isc-dhcp-server`

If using dhcpd, the following example configuration will add a network (192.168.100.0/24) for servicing DHCP requests for ZTPServer:

```
subnet 192.168.100.0 netmask 255.255.255.0 {
  range 192.168.100.200 192.168.100.205;
  option routers 192.168.100.1;
  option domain-name-servers <ipaddr>;
  option domain-name "<org>";
  option bootfile-name "http://<ztp_hostname_or_ip>:<port>/bootstrap";
}
```

Enable and start the dhcpd service.

RedHat (and derivative Linux implementations) `bash-3.2# sudo /usr/bin/systemctl enable dhcpd.service` `bash-3.2# sudo /usr/bin/systemctl start dhcpd.service`

Ubuntu (and derivative Linux implementations) `bash-3.2# sudo /usr/sbin/service isc-dhcp-server start`

Check that /etc/init/isc-dhcp-server.conf is configured for automatic startup on boot.

Edit the global configuration file located at `/etc/ztpserver/ztpserver.conf` (if needed). See the *Global configuration file* options for more information.

Now, you are ready to *Startup* ZTPServer.

2.3 Startup

HTTP Server Deployment Options

ZTPServer is a Python WSGI compliant application that can be deployed behind any WSGI web server or run as a standalone application. This section provides details for configuring ZTPServer to run under various WSGI compliant web servers. By default, ZTPServer ships with a single-threaded server that is sufficient for testing.

2.3.1 Standalone server

To start the standalone ZTPServer, exec the ztps binary

```
[root@ztpserver ztpserver]# ztps
INFO: [app:115] Logging started for ztpserver
INFO: [app:116] Using repository /usr/share/ztpserver
Starting server on http://<ip_address>:<port>
```

The following options may be specified when starting the ztps binary:

```
-h, --help            show this help message and exit
--version, -v         Displays the version information
--conf CONF, -c CONF  Specifies the configuration file to use
--validate FILENAME   Runs a validation check on neighbordb
--debug               Enables debug output to the STDOUT
```

When ZTPServer starts, it reads the path information to neighbordb and other files from the global configuration file. Assuming that the DHCP server is serving DHCP offers which include the path to the ZTPServer bootstrap script in Option 67 and that the EOS nodes can access the bootstrap file over the network, the provisioning process should now be able to automatically start for all the nodes with no startup configuration.

2.3.2 Apache (mod_wsgi)

If using Apache, this section provides instructions for setting up ZTPServer using mod_wsgi. This section assumes the reader is familiar with Apache and has already installed mod_wsgi. For details on how to install mod_wsgi, please see the modwsgi Quick Installation Guide.

To enable ZTPServer for an Apache server, we need to add the following WSGI configuration (example)

```
LoadModule wsgi_module modules/mod_wsgi.so

WSGIDaemonProcess ztpserver user=www-data group=www-data threads=5
WSGIScriptAlias / /etc/ztpserver/ztpserver.wsgi

<Directory /ztpserver>
    WSGIProcessGroup ztpserver
    WSGIApplicationGroup %{GLOBAL}
    Order deny,allow
    Allow from all
</Directory>
```

WSGIScriptAlias should point to the ztpserver.wsgi file which is installed by default under /etc/ztpserver. The `<Directory /ztpserver>` tag assigns the path prefix for the ZTPServer url. The ZTPServer configuration must be updated to include the URL path prefix (`/ztpserver` in this example).

To update the ZTPServer configuration, edit the default configuration file found at /etc/ztpserver/ztpserver.conf by modifying or adding the following line under the [default] section:

```
server_url = http://192.168.1.34/ztpserver
```

where /ztpserver is the same name as the directory entry configured above. Once completed, restart Apache and you should now be able to access your ZTPServer at the specified URL. To test, simply use curl - for example:

```
curl http://1921.68.1.34/ztpserver/bootstrap
```

If everything is configured properly, curl should be able to retrieve the bootstrap script. If there is a problem, all of the ZTPServer log messages should be available under the Apache server error logs.

2.4 Configuration

The ZTPServer uses a series of YAML files to provide its various configuration and databases. Use of the YAML format makes the file easier to read and makes it easier and more intuitive to add/update entries (as opposed to other files formats such as JSON, or binary formats such as SQL).

2.4.1 Configuration Types

There are 2 general types of configurations supported by ZTPServer, Static and Dynamic provisioning.

Static provisioning:

Manually create node entries in /nodes and a startup-configuration. In order to do that:

- Create a new directory for each node under [data_root]/nodes, using the system unique_id as the name.
- Place a startup-config in the newly-created folder.

Example:

```
[root@localhost ztpserver]# mkdir /usr/share/ztpserver/nodes/000c29f3a39g
[root@localhost ztpserver]# cp myconfig /usr/share/ztpserver/nodes/000c29f3a39g/startup-config
```

Topology validation is still an active component of a static provisioning configuration at defaults. This allows a customer to validate cabling even with a statically defined node. If `disable_topology_validation = true` in `/etc/ztpserver/ztpserver.conf` then you won't need to create a pattern file in the directory for topology

validation, if it is set to "false" (default), then you'll need to place a "pattern" file in the specific node directory, using a similar syntax as neighbordb.

e.g.: `/usr/share/ztpserver/nodes/ABC12345678/pattern`

This can be as simple as below, but must exist. See the *Static neighbordb and /node/<unique-id>/pattern file* example.

```
name: static_node
interfaces:
- any: any:any
```

Dynamic provisioning:

This method assumes that you do not create a node entry for each node manually. Instead create a neighbordb entry with at least one pattern that maps to a definition. This requires editing: /usr/share/ztpserver/neighbordb

And creating at least one pattern. See the *Dynamic neighbordb or pattern file* example.

Once you've created the neighbordb entry, you'll need to match a definition file placed in: /usr/share/ztpserver/definitions/

See the *Sample dynamic definition file* example.

The combination of a neighbordb match and a template definition with dynamic resource allocation allow the same definition to be used for multiple nodes.

2.4.2 Global configuration

Global configuration file

The global ZTPServer configuration file can be found at `/etc/ztpserver/ztpserver.conf`. It uses INI format. (For format details, see top section Python configparser).

An alternate location for the global configuration file may be specified by using the `--conf` command line option: e.g.

```
(bash)# ztps --help
usage: ztpserver [options]

optional arguments:
  -h, --help            show this help message and exit
  --version, -v         Displays the version information
  --conf CONF, -c CONF  Specifies the configuration file to use
  --validate FILENAME   Runs a validation check on neighbordb
  --debug               Enables debug output to the STDOUT
(bash)# ztps --conf /var/ztps.conf
```

If the global configuration file is updated, the server must be restarted in order to pick up the new configuration.

Sections and attributes

```
[default]

# Location of all ztps boostrap process data files
# default=/var/lib/ztpserver
data_root=<PATH>
```

```
# UID used in the /nodes structure (serialnum is not supported yet)
# default=serialnum
identifier=<serialnum | systemmac>

# Server URL to-be-advertised to clients (via POST replies) during the bootstrap process
# default=http://ztpserver:8080
server_url=<URL>

# Enable local logging
# default=True
logging=<True | False>

# Enable console logging
# default=True
console_logging=<True | False>

# Globally disable topology validation in the bootstrap process
# default=False
disable_topology_validation=<True | False>

[server]
# Note: this section only applies to using the standalone server.  If
# running under a WSGI server, these values are ignored

# Interface to which the server will bind to (0:0:0:0 will bind to
# all available IPv4 addresses on the local machine)
# default=0.0.0.0
interface=<IP addr>

# TCP listening port
# default=8080
port=<TCP port>

[files]
# Path for the files directory (overriding data_root/files)
# default=files
folder=<path>
# default=data_root (from above)
path_prefix=<path>

[actions]
# Path for the actions directory (overriding data_root/actions)
# default=actions
folder=<path>
# default=data_root (from above)
path_prefix=<path>

[bootstrap]
# Path for the bootstrap directory (overriding data_root/bootstrap)
# default=bootstrap
folder=<path>
# default=data_root (from above)
path_prefix=<path>

# Bootstrap filename
# default=bootstrap
filename=<name>
```

```
[neighbordb]
# Neighbordb filename (file located in data_root)
# default=neighbordb
filename=<name>
```

Environment Variables in the global configuration

Note: Configuration values may be overridden by setting environment variables, if the configuration attribute supports it. This is mainly used for testing and should not be used in production deployments.

Configuration values that support environment overrides use the `environ` keyword, as shown below:

```
runtime.add_attribute(StrAttr(
    name='data_root',
    default='/usr/share/ztpserver',
    environ='ZTPS_DEFAULT_DATAROOT'
))
```

In the above example, the `data_root` value is normally configured in the [default] section as `data_root`; however, if the environment variable `ZTPS_DEFAULT_DATAROOT` is defined, it will take precedence.

Data Directory Structure

The ZTPServer side components are housed in a single directory defined by the `data_root` variable in the global configuration file. The directory location will vary depending on the configuration in `/etc/ztpserver/ztperserver.conf`. The data_root is loaded when ztps is executed. The following directory structure is normally used:

```
[data_root]
    bootstrap/
        bootstrap
        bootstrap.conf
    nodes/
        <unique_id)>/
            startup-config
            definition
            pattern
            .node
            attributes
    actions/
    files/
    definitions/
    resources/
    neighbordb
```

Bootstrap configuration

`[data_root]/bootstrap/` contains files that control the bootstrap process on a node.

- **bootstrap** is the base bootstrap script which is going to be served to all clients in order to start and run the bootstrap process. Before serving the script to the clients, the server replaces any references to $SERVER with the value of `server_url` in the global configuration file

- **bootstrap.conf** is a configuration file which defines the local logging configuration on the nodes (during the bootstrap process). The file is loaded on each request.

  e.g.

```
---
logging:
  -
    destination: "ztps.ztps-test.com:514"
    level: DEBUG
  - destination: file:/tmp/ztps-log
    level: DEBUG
  - destination: ztps-server:1234
    level: CRITICAL
  - destination: 10.0.1.1:9000
    level: CRITICAL
xmpp:
  domain: im.ztps-test.com
  username: bootstrap
  password: eosplus
  rooms:
    - ztps
    - ztps-room2
```

2.4.3 Node-specific configuration

`[data_root]/nodes/` contains node-specific configuration files.

Startup configuration

`startup-config` provides a static startup configuration file. If this file is present in a node's folder, when the node sends a GET request to `/nodes/<unique_id>` where unique_id is either the serial number or system-mac, the server will respond with a static definition that includes:

- a **replace_config** action which will install the configuration file on the switch (see actions section below for more on this)
- all the **actions** from the local **definition** file (see definition section below for more on this) which have the `always_execute` attribute set to `True`

Definition file

The **definition** file is the collection of actions which are going to be performed during the bootstrap process for the node. The definition file can be either: **manually created** OR **auto-generated by the server** when the node matches one of the patterns in **neighbordb**. The definition file is generated based on the definition file associated with the matching pattern in **neighbordb**.

```
name: <system name>

actions:
    - name: <name>
    action: <action name>

    attributes:                     # attributes at action scope
        always_execute: True        # optional, default False
        <key>: <value>
```

```
        <key>: <value>

    onstart:   <msg>                # message to log before action is executed
    onsuccess: <msg>                # message to log if action execution succeeds
    onfailure: <msg>                # message to log if action execution fails

attributes:                         # attributes at global scope
    <key>: <value>
    <key>: <value>
    <key>: <value>
```

Attributes

Attributes are either key/value pairs, key/dictionary pairs, key/list pairs or key/reference pairs. They are all sent to the client in order to be passed in as arguments to actions.

key/reference pairs are evaluated before being sent to the client.

Here are a few examples:

- key/value:

```
attributes:
    my_attribute : my_value
```

- key/dictionary

```
attributes:
    my_dict_attribute:
        key1: value1
        key2: value2
```

- key/list:

```
attributes:
    - my_value1
    - my_value2
    - my_valueN
```

- key/reference:

```
attributes:
    my_attribute : $my_other_attribute
```

key/reference attributes are identified by the fact that the value starts with the '$' sign, followed by the name of another attribute. They are evaluated before being sent to the client

Example:

```
attributes:
    my_other_attribute: dummy
    my_attribute : $my_other_attribute
```

will be evaluated to:

```
attributes:
    my_other_attribute: dummy
    my_attribute : dummy
```

If a reference points to a non-existing attribute, then the variable substitution will result in a value of *None*.

Note: For release 1.0, only **one level of indirection** is allowed - if multiple levels of indirection are used, then the data sent to the client will contain unevaluated key/reference pairs in the attributes list (which might lead to failures or unexpected results in the client).

The values of the attributes can be either strings, lists, dictionaries, references to other attributes or functions.

The supported functions are:

- **allocate(resource_pool)** - allocate available resource from resource pool; the allocation is perform on the server side and the result of the allocation is passed to the client via the definition

---

Note: Functions can only be used with strings as arguments, currently. See section on add_config action for examples.

Attributes can be defined in three places:

- in the node's attributes file (see below)
- in the definition, at global scope
- in the definition, at action scope

For key/value, key/list and and key/reference attributes, in case of conflicts between the three scopes, the following order of precidence rules are applied to determine the final value to send to the client:

1. action scope in the definition takes precedence
2. attributes file comes next
3. global scope in the definition comes last

For key/dict attributes, in case of conflicts between the scopes, the dictionaries are merged. In the event of dictionary key conflicts, the same precidence rules from above apply.

Pattern file

The **pattern** file provides a *statically typed* pattern match which is used to validate the node's neighbors during the bootstrap process (if topology validation is enabled). The pattern file can be either:

- manually created
- auto-generated by the server, when the node matches one of the patterns in `neighbordb`. The pattern that is matched in `neighbordb` is, then, written to this file and used for topology validation in subsequent re-runs of the bootstrap process.

The format of a pattern is very similar to the format of `neighordb` (see neighbordb section below):

```
variables:
    <variable_name>: <function>
...

name: <single line description of pattern>
definition: <defintion_url>
interfaces:
    - <port_name>:<system_name>:<neighbor_port_name>:<tags>
    - <port_name>:
        device: <system_name>
        port: <neighbor_port_name>
        tags: <comma delimited tags list>
...
```

If the pattern file is missing when the node makes a GET request for its definition, the server will log a message and return either:

- 400 (BAD_REQUEST) if topology validation is enabled
- 200 (OK) if topology validation is disabled

If topology validation is enabled, the following pattern can be used in order to disable it locally for a node (the pattern from below will match **any** node):

```
name: <pattern name>
interfaces:
    - any: any:any
```

Node details

The `.node` file contains a cached copy of the node's details that were received during the POST request the node makes to `/nodes (URI)`. This cache is used to validate the node's neighbors against the `pattern` file, if topology validation is enabled (during the GET request the node makes in order to retrieve its definition).

Attributes file

`attributes` is a file which can be used in order to store attributes associated with the node's definition. This is especially useful whenever multiple nodes share the same definition - in that case, instead of having to edit each node's definition in order to add the attributes (at the global or action scope), all nodes can share the same definition (which might be symlinked to their individual node folder) and the user only has to create the attributes file for each node. The `attributes` file should be a valid key/value YAML file.

2.4.4 Actions

`[data_root]/actions/` contains all of the actions available for use in definitions. More details about each action can be found at the top of the corresponding Python file.

Action	Description	Required Attributes
`add_config` *	Adds a section of config to the final startup-config file	url
`copy_file`	Copies a file from the server to the destination node	src_url, dst_url, overwrite, mode
`install_cli_plugin`	Installs a new EOS CLI plugin and configures rc.eos	url
`install_extension`	Installs a new EOS extension	extension_url, autoload, force
`install_image`	Validates and installs a specific version of EOS	url, version
`replace_config`	Sends an entire startup-config to the node (overrides (overrides add_config)	url
`send_email`	Sends an email to a set of recipients routed through a relay host. Can include file attachments	smarthost, sender, receivers, subject, body, attachments, commands

Additional details on each action are available in the *Actions* module docs.

Note:

- The 'add_config' action supports applying block of EOS configuration commands to a node's startup-config.

---

e.g.

Let's assume that we have a block of configuration that adds a list of NTP servers to the startup configuration. The action would be constructed as such:

```
actions:
    - name: configure NTP
      action: add_config
      attributes:
        url: /files/templates/ntp.template
```

The above action would reference the `ntp.template` file which would configure NTP. The template file could look like the one from below:

```
ntp server 0.north-america.pool.ntp.org
ntp server 1.north-america.pool.ntp.org
ntp server 2.north-america.pool.ntp.org
ntp server 3.north-america.pool.ntp.org
```

When this action is called, the configuration snippet above will be appended to the `startup-config` file.

The configuration templates can also contains **variables**, which are automatically substituted during the action's execution. A variable is marked in the template via the '$' symbol.

e.g. Let's assume a need for a more generalized template that only needs node specific values changed (such as a hostname and management IP address). In this case, we'll build an action that allows for **variable substitution** as follows.

```
actions:
    - name: configure system
      action: add_config
      attributes:
        url: /files/templates/system.template
        variables:
            hostname: veos01
            ipaddress: 192.168.1.16/24
```

The corresponding template file `system.template` will provide the configuration block:

```
hostname $hostname
!
interface Management1
    description OOB interface
    ip address $ipaddress
    no shutdown
```

This will result in the following configuration being added to the `startup-config`:

```
hostname veos01
!
interface Management1
    description OOB interface
    ip address 192.168.1.16/24
    no shutdown
```

Note that in each of the examples, above, the template files are just standard EOS configuration blocks.

2.4.5 Resources

`[data_root]/files/` contains the files that actions might request from the server. For example, `[data_root]/files/images/` could contain all EOS SWI files.

2.4.6 Definitions

`[data_root]/definitions/` contains a set of shared definition files which can be associated with pattern in neighbordb (see the *Neighbordb* section below) or symlink-ed from nodes' folders.

2.4.7 Resource pools

`[data_root]/resources/` contains global resource pools from which attributes in definitions can be allocated via the allocate(...) function.

The resource pools provide a way to dynamically allocate a resource to a node when the node definition is created. The resource pools are key/value YAML files that contain a set of resources to be allocated to a node (whenever the allocate(...) function is used in the definition).

In the example below, a resource pool contains a series of 8 IP addresses to be allocated. Entries which are not yet allocated to a node are marked using the `null` descriptor.

```
192.168.1.1/24: null
192.168.1.2/24: null
192.168.1.3/24: null
192.168.1.4/24: null
192.168.1.5/24: null
192.168.1.6/24: null
192.168.1.7/24: null
192.168.1.8/24: null
```

When a resource is allocated to a node's definition, the first available null value will be replaced by the node's unique_id. Here is an example:

```
192.168.1.1/24: 001c731a2b3c
192.168.1.2/24: null
192.168.1.3/24: null
192.168.1.4/24: null
192.168.1.5/24: null
192.168.1.6/24: null
192.168.1.7/24: null
192.168.1.8/24: null
```

On subsequent attempts to allocate the resource to the same node, ZTPS will first check to see whether the node has already been allocated a resource from the pool. If it has, it will reuse the resource instead of allocating a new one.

In order to free a resource from a pool, simply turn the value associated to it back to `null`, by editing the resource file.

2.4.8 Neighbordb

The `neighbordb` YAML file defines mappings between node descriptions and node definitions. If a node does not already have a node definition, then the node's details are attempted to be matched against the patterns in `neighbordb`. If a match is successful, then a node definition will be automatically generated for the node.

```
variables:
    variable_name: function
...
patterns*:
    - name*: <single line description of pattern>
      definition*: <defintion_url>
      node: <unique_id>
```

```
      variables:
        <variable_name>: <function>
      interfaces*:
        - <port_name>*: <system_name>*:<neighbor_port_name>:<tags>
        - <port_name>*:
            device*: <system_name>*
            port: <neighbor_port_name>
            tags: <comma delimited tags list>
...
```

Note: Itemsmarked wiht * are mandatory elements. Everything else is optional.

variables

This section allows for the definition of variables in neighbordb. The variables can be used to match remote device and/or interface names (`<system_name>`, `<neighbor_port_name>` above) of a node during the pattern matching stage. The supported values are:

string same as exact(string) from below

exact (pattern) defines a pattern that must be matched exactly (Note: this is the default function if another function is not specified)

regex (pattern) defines a regex pattern to match the node name against

includes (string) defines a string that must be present in the node name

excludes (string) defines a string that must not be present in the node name

itentifier

System serial number or MAC address of a node, depending on the global 'identifier' setting in ztpserver.conf.

port_name

Local node interface - supported values (MUST start with **"Ethernet"**, if not keyword):

- **Any interface**
  - any
- **No interface**
  - none
- **Explicit interface**
  - Ethernet1
  - Ethernet2/4
- **Interface list/range**
  - Ethernet1-2
  - Ethernet1,3
  - Ethernet1-2,3/4

  - Ethernet1-2,4
  - Ethernet1-2,4,6
  - Ethernet1-2,4,6,8-9
  - Ethernet4,6,8-9
  - Ethernet10-20
  - Ethernet1/3-2/4 *
  - Ethernet3-$ *
  - Ethernet1/10-$ *
- **All Interfaces on a Module**
  - Ethernet1/$ *

Note: * Available in future releases.

---

system_name:neighbor_port_name

Remote system and interfaces - supported values (STRING = any string which does not contain any white spaces):

- `any`: interface is connected
- `none`: interface is NOT connected
- `<STRING>:<STRING>`: interface is connected to specific device/interface
- `<STRING>` (Note: if only the device is configured, then 'any' is implied for the interface. This is equal to `<DEVICE>:any`): interface is connected to device
- `<DEVICE>:any`: interface is connected to device
- `<DEVICE>:none`: interface is NOT connected to device (might be connected or not to some other device)
- `$<VARIABLE>:<STRING>`: interface is connected to specific device/interface
- `<STRING>:<$VARIABLE>`: interface is connected to specific device/interface
- `$<VARIABLE>:<$VARIABLE>`: interface is connected to specific device/interface
- `$<VARIABLE>` ('any' is implied for the interface. This is equal to `$<VARIABLE>:any`): interface is connected to device
- `$<VARIABLE>:any`: interface is connected to device
- `$<VARIABLE>:none`: interface is NOT connected to device (might be connected or not to some other device)

port_name: system_name:neighbor_port_name

Negative constraints

1. `any:  DEVICE:none`: no port is connected to DEVICE
2. `none:  DEVICE:any`: same as above
3. `none:  DEVICE:none`: same as above
4. `none:  any:PORT`: no device is connected to PORT on any device

5. `none: DEVICE:PORT`: no device is connected to DEVICE:PORT
6. `INTERFACES: any:none`: interfaces not connected
7. `INTERFACES: none:any`: same as above
8. `INTERFACES: none:none`: same as above
9. `INTERFACES: none:PORT`: interfaces not connected to PORT on any device
10. `INTERFACES: DEVICE:none`: interfaces not connected to DEVICE
11. `any: any:none`: bogus, will prevent pattern from matching anything
12. `any: none:none`: bogus, will prevent pattern from matching anything
13. `any: none:any`: bogus, will prevent pattern from matching anything
14. `any: none:PORT`: bogus, will prevent pattern from matching anything
15. `none: any:any`: bogus, will prevent pattern from matching anything
16. `none: any:none`: bogus, will prevent pattern from matching anything
17. `none: none:any`: bogus, will prevent pattern from matching anything
18. `none: none:none`: bogus, will prevent pattern from matching anything
19. `none: none:PORT`: bogus, will prevent pattern from matching anything

Positive constraints

1. `any: any:any`: matches anything
2. `any: any:PORT`: matches any interface connected to any device's PORT
3. `any: DEVICE:any`: matches any interface connected to DEVICE
4. `any: DEVICE:PORT`: matches any interface connected to DEVICE:PORT
5. `INTERFACES: any:any`: matches if local interfaces is one of INTERFACES
6. `INTERFACES: any:PORT`: matches if one of INTERFACES is connected to any device's PORT
7. `INTERFACES: DEVICE:any`: matches if one of INTERFACES is connected to DEVICE
8. `INTERFACES: DEVICE:PORT`: matches if one of INTERFACES is connected to DEVICE:PORT

tags

Supported in future releases.

2.5 Examples

2.5.1 Global configuration file

```
[default]
# Location of all ztps boostrap process data files
data_root = /usr/share/ztpserver

# UID used in the /nodes structure (serialnumber or systemmac)
identifier = serialnumber

# Server URL to-be-advertised to clients (via POST replies) during the bootstrap process
server_url = http://172.16.130.10:8080

# Enable local logging
logging = True

# Enable console logging
console_logging = True

# Globally disable topology validation in the bootstrap process
disable_topology_validation = False

[server]
# Note: this section only applies to using the standalone server.  If
# running under a WSGI server, these values are ignored

# Interface to which the server will bind to (0:0:0:0 will bind to
# all available IPv4 addresses on the local machine)
interface = 172.16.130.10

# TCP listening port
port = 8080

[ files]
# Path for the files directory (overriding data_root/files)
folder = files
path_prefix = /usr/share/ztpserver

[actions]
# Path for the actions directory (overriding data_root/actions)
folder = actions
path_prefix = /usr/share/ztpserver
```

```
[bootstrap]
# Path for the bootstrap directory (overriding data_root/bootstrap)
folder = bootstrap
path_prefix = /usr/share/ztpserver

# Bootstrap filename
filename = bootstrap

[neighbordb]

# Neighbordb filename (file located in data_root)
filename = neighbordb
```

2.5.2 Dynamic neighbordb or pattern file

```
---
patterns:
#dynamic sample
  - name: dynamic_sample
    definition: tor1
    interfaces:
      - Ethernet1: spine1:Ethernet1
      - Ethernet2: spine2:Ethernet1
      - any: ztpserver:any

  - name: dynamic_sample2
    definition: tor2
    interfaces:
      - Ethernet1: spine1:Ethernet2
      - Ethernet2: spine2:Ethernet2
      - any: ztpserver:any
```

2.5.3 Static neighbordb and /node/<unique-id>/pattern file

```
---
patterns:
#static sample
  - name: static_node
    node: 000c29f3a39g
    interfaces:
      - any: any:any
```

2.5.4 Sample dynamic definition file

```
---
actions:
  -
    action: install_image
    always_execute: true
    attributes:
      url: files/images/vEOS.swi
      version: 4.13.5F
    name: "validate image"
```

```
  -
    action: add_config
    attributes:
      url: files/templates/ma1.template
      variables:
        ipaddress: allocate('mgmt_subnet')
    name: "configure ma1"
  -
    action: add_config
    attributes:
      url: files/templates/system.template
      variables:
        hostname: allocate('tor_hostnames')
    name: "configure global system"
  -
    action: add_config
    attributes:
      url: files/templates/login.template
    name: "configure auth"
  -
    action: add_config
    attributes:
      url: files/templates/ztpprep.template
    name: "configure ztpprep alias"
  -
    action: add_config
    attributes:
      url: files/templates/snmp.template
      variables: $variables
    name: "configure snmpserver"
  -
    action: add_config
    attributes:
      url: files/templates/configpush.template
      variables: $variables
    name: "configure config push to server"
  -
    action: copy_file
    always_execute: true
    attributes:
      dst_url: /mnt/flash/
      mode: 777
      overwrite: if-missing
      src_url: files/automate/ztpprep
    name: "automate reload"
attributes:
  variables:
    ztpserver: 172.16.130.10
name: tora
```

2.5.5 Sample templates

```
#login.template
#::::::::::::::
username admin priv 15 secret admin
```

```
#ma1.template
#:::::::::::::
interface Management1
  ip address $ipaddress
  no shutdown

#hostname.template
#:::::::::::::
hostname $hostname
```

2.5.6 Sample resources

```
#mgmt_subnet
#:::::::::::::::
192.168.100.210/24: null
192.168.100.211/24: null
192.168.100.212/24: null
192.168.100.213/24: null
192.168.100.214/24: null

#tor_hostnames
#:::::::::::::::
veos-dc1-pod1-tor1: null
veos-dc1-pod1-tor2: null
veos-dc1-pod1-tor3: null
veos-dc1-pod1-tor4: null
veos-dc1-pod1-tor5: null
```

2.5.7 Mode Examples

Example #1: strongly typed definition with a strongly typed map

```
---
- name: standard leaf definition
  definition: leaf_template
  node: ABC12345678
  interfaces:
    - Ethernet49: pod1-spine1:Ethernet1/1
    - Ethernet50:
        device: pod1-spine2
        port: Ethernet1/1
```

In example #1, the topology map would only apply to a node with serial number, the default ID, equal to **ABC12345678**. The following interface map rules apply:

- Interface Ethernet49 must be connected to node pod1-spine1 on port Ethernet1/1
- Interface Ethernet50 must be connected to node pod1-spine2 on port Ethernet1/1

Example #2: strongly typed definition with loose typed map

```
---
- name: standard leaf definition
  definition: leaf_template
```

```
  node: 001c73aabbcc
  interfaces:
    - any: regex('pod\d+-spine\d+'):Ethernet1/$
    - any:
        device: regex('pod\d+-spine1')
        port: Ethernet2/3
```

In this example, the topology map would only apply to the node with system mac address equal to **001c73aabbcc**. This requires that identifier be set to systemmac in the global ztpserver.conf file. The following interface map rules apply:

- Any interface must be connected to node that matches the regular expression 'pod+-spine+' on port Ethernet1/$ (any port on module 1)
- Any interface and not the interface selected in the previous step must be connected to a node that matches the regular expression 'pod+-spine1' and is connected on port Ethernet2/3

Example #3: loose typed definition with a loose typed map

```
---
- name: standard leaf definition
  definition: dc-1/pod-1/leaf_template
  variables:
    - not_spine: excludes('spine')
    - any_spine: regex('spine\d+')
    - any_pod: includes('pod')
    - any_pod_spine: any_spine and any_pod*
  interfaces:
    - Ethernet1: $any_spine:Ethernet1/$
    - Ethernet2: $pod1-spine2:any
    - any: excludes('spine1'):Ethernet49
    - any: excludes('spine2'):Ethernet49
    - Ethernet49:
        device: $not_spine
        port: Ethernet49
    - Ethernet50:
        device: excludes('spine')
        port: Ethernet50
```

Note: * In a future release.

This example pattern could apply to any node that matches the interface map. In includes the use of variables for cleaner implementation and pattern re-use.

- Variable not_spine matches any node name where 'spine' doesn't appear in the string
- Variable any_spine matches any node name where the regular expression 'spine+' matches the name
- Variable any_pod matches any node name where that includes the name 'pod' in it
- **Variable any_pod_spine combines variables any_spine and any_pod into a complex variable that includes any name that matches the regular express 'spine+' and the name includes 'pod' (not yet supported)**
- Interface Ethernet1 must be connected to a node that matches the any_spine pattern and is connected on Ethernet1/$ (any port on module 1)
- Interface Ethernet2 must be connected to node 'pod1-spine2' on any Ethernet port
- Interface any must be connected to any node that doesn't have 'spine1' in the name and is connected on Ethernet49

- Interface any must be connected to any node that doesn't have 'spine2' in the name and wasn't already used and is connected to Ethernet49
- Interface Ethernet49 matches if it is connected to any node that matches the not_spine pattern and is connected on port 49
- Interface Ethernet50 matches if the node is connected to port Ethernet50 on any node whose name does not contain 'spine'

Example #4: loosely typed definition with loosely typed map

```
---
- name: sample mlag definition
  definition: mlag_leaf_template
  variables:
    any_spine: includes('spine')
    not_spine: excludes('spine')
  interfaces:
    - Ethernet1: $any_spine:Ethernet1/$
    - Ethernet2: $any_spine:any
- Ethernet3: none
- Ethernet4: any
- Ethernet5:
    device: includes('oob')
    port: any
- Ethernet49: $not_spine:Ethernet49
- Ethernet50: $not_spine:Ethernet50
```

This is a similar example to #3 that demonstrates how an MLAG pattern might work.

- Variable any_spine defines a pattern that includes the word 'spine' in the name
- Variable not_spine defines a pattern that matches the inverse of any_spine
- Interface Ethernet1 matches if it is connected to any_spine on port Ethernet1/$ (any port on module 1)
- Interface Ethernet2 matches if it is connected to any_spine on any port
- Interface 3 matches so long as there is nothing attached to it
- Interface 4 matches so long as something is attached to it
- Interface 5 matches if the node contains 'oob' in the name and is connected on any port
- Interface49 matches if it is connected to any device that doesn't have 'spine' in the name and is connected on Ethernet50
- Interface50 matches if it is connected to any device that doesn't have 'spine' in the name and is connected on port Ethernet50

2.5.8 More examples

Additional ZTPServer file examples are available on GitHub at the ZTPServer Demo.

2.6 Tips and tricks

2.6.1 How do I update my local copy of ZTPServer from GitHub?

Script

Go to the ZTPS directory where you previously cloned the GitHub repository in order to pull the latest code and execute:

```
/utils/refresh_ztps
```

Manually

Remove the existing ZTPS files:

```
rm -rf /usr/share/ztpserver/actions/*
rm -rf /usr/share/ztpserver/bootstrap/*
rm -rf /usr/lib/python2.7/site-packages/ztpserver*
rm -rf /bin/ztps*
rm -rf /home/ztpuser/ztpserver/ztpserver.egg-info/
rm -rf /home/ztpuser/ztpserver/build/*
```

Go to the ZTPS directory where you previously cloned the GitHub repository in order to pull the latest code, build an install it:

```
bash-3.2$ git pull
bash-3.2$ python setup.py build
bash-3.2$ python setup.py install
```

2.6.2 My server keeps failing to load my resource files. What's going on?

Did you know?

```
a:b is INVALID YAML
a: b is VALID YAML options
```

Check out YAML syntax checker for more.

2.6.3 How can I test ZTPS without having to reboot the switch every time?

From a bash shell:

```
# retrieve the bootstrap file from server
wget (http://<ZTPS_SERVER>:<PORT>/bootstrap
# make file executable
sudo chmod 777 bootstrap
```

```
# execute file
sudo ./bootstrap
```

2.6.4 How do I override the default system-mac in vEOS?

Add the desired MAC address to the first line of the file /mnt/flash/system_mac_address, then reboot

```
[admin@localhost ~]$ echo 1122.3344.5566 > /mnt/flash/system_mac_address
```

2.6.5 How do I override the default serial number or system-mac in vEOS?

As of vEOS 4.14.0, the serial number and system mac address can be configured with a file in /mnt/flash/veos-config. After modifying SERIALNUMBER or SYSTEMMACADDR, a reboot is required for the changes to take effect.

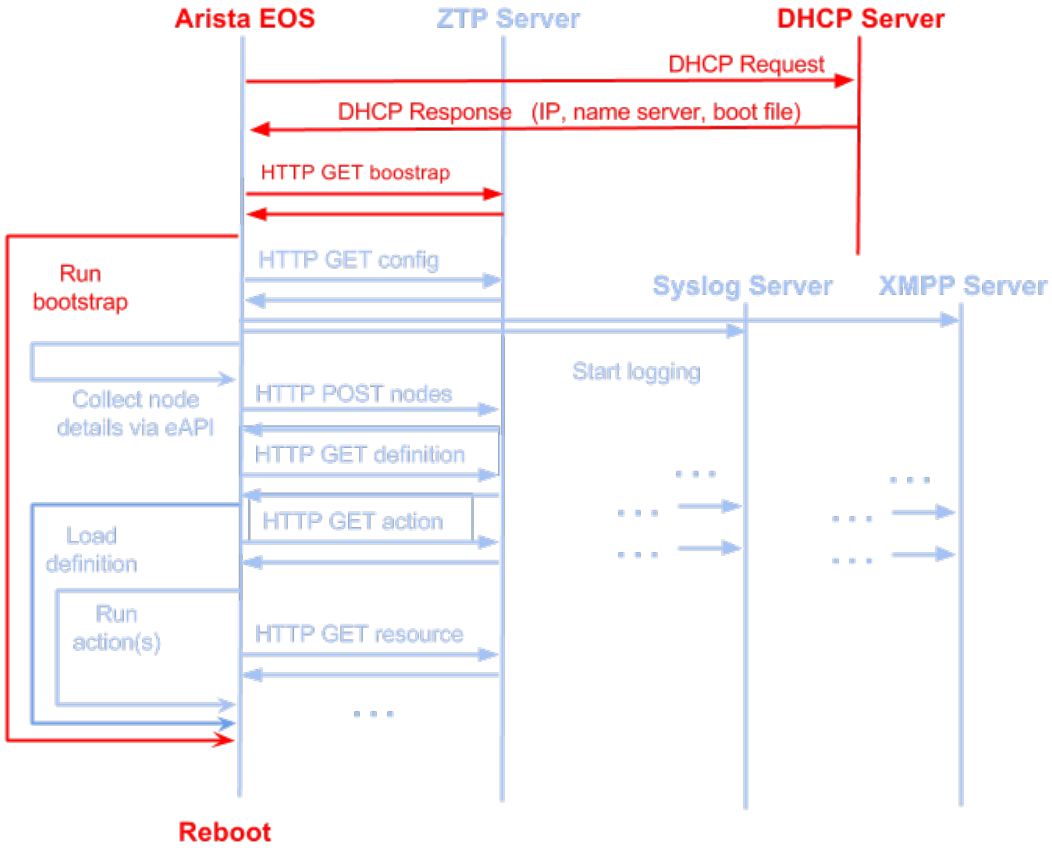
```
SERIALNUMBER=ABC12345678
SYSTEMMACADDR=1122.3344.5566
```

2.7 Internals

2.7.1 Implementation Details

Client-Server Message Flows

A high level view of the client - server message flows can be seen in the following diagram:



Note: Red lines indicate Arista EOS (ZTP) flows.

Blue lines indicate ZTPServer's bootstrap client's flows.

Client-side implementation details

Action attributes

The bootstrap script will pass in as argument to the main method of each action a special object called 'attributes'. The only API the action needs to be aware for this object is the 'get' method, which will return the value of an attribute, as configured on the server:

- the value can be local to a particular action or global
- if an attribute is defined at both the local and global scopes, the local value takes priority
- if an attribute is not defined at either the local or global level, then the 'get' method will return **None**

e.g. (action code)

```
def main(attributes):
    print attributes.get('software_image')
```

Besides the values coming from the server, a couple of **special entries*** (always upper case) are also contained in the attributes object:

1. 'NODE': a node object for making eAPI calls to localhost See the *Bootstrap Client* documentation.

Note: Object has other functionality as well and more of it could be documented and exposed in the future - this is the only one interesting for now.*

---

e.g. (action_code)

```
def main(attributes):
    print attributes.get('NODE').api_enable_cmds(['show version'])
```

Bootstrap URLs

1. DHCP response contains the **URL pointing to the bootstrap script**
2. The location of the bootstrap configuration server is hardcoded in the bootstrap script, using the SERVER global variable. The bootstrap script uses this base address in order to generate the **URL to use in order to GET the logging details**: `BASE_URL/config` e.g.

   ```
   SERVER = 'http://my-bootstrap-server'    # Note that the transport mechanism is
                                            # included in the URL
   ```

3. The bootstrap script uses the SERVER base address in order to compute the **URL to use in order to POST the node's information:** `BASE_URL/config`
4. The bootstrap script uses the 'location' header in the POST reply as the **URL to use in order to request the definition**
5. **Actions and resources URLs**& are computed by using the base address in the bootstrap script: BASE_URL/actions/, BASE_URL/files/

Note: In future releases, the definition will contain an extra optional attribute for each action/resource which could be used in order to redirect the bootstrap client to another server in order to retrieve that resource. This will enable a more distributed model for serving ZTP actions and resources.*

2.7.2 Client - Server API

URL Endpoints

HTTP Method	URI
GET	/bootstrap/config
GET	/bootstrap
POST	/nodes
PUT	/nodes/{id}
GET	/nodes/{id}
GET	/actions/{name}
GET	/files/{filepath}

GET bootstrap script

GET /bootstrap
Returns the default bootstrap script

Response

```
Status: 200 OK
Content-Type: text/x-python
```

Note: For every request, the bootstrap controller on the ZTPServer will attempt to perform the following string replacement in the bootstrap script): **"$SERVER" —> the value of the "server_url" variable in the server's configuration file** This string-replacement will point the bootstrap client back to the server, in order to enable it to make additional requests for further resources.

- if the `server_url` variable is missing in the server's configuration file, 'http://ztpserver:8080' is used by default
- if the `$SERVER` string does not exist in the bootstrap script, the controller will log a warning message and continue

GET logging configuration

GET /bootstrap/config
Returns the logging configuration from the server.

Request

```
GET /bootstrap/config HTTP/1.1
Host:
Accept:
Content-Type: text/html
```

Response

```
Status: 200 OK
Content-Type: application/json
{
    "logging"*: [ {
        "destination": "file:/<PATH>" | "<HOSTNAME OR IP>:<PORT>",    //localhost enabled
                                                                      //by default
        "level"*:        <DEBUG | CRITICAL | ...>,
    } ]
},
```

```
    "xmpp"*:{
        "server":           <IP or HOSTNAME>,
        "port":             <PORT>,                 // Optional, default 5222
        "username"*:        <USERNAME>,
        "domain"*:          <DOMAIN>,
        "password"*:        <PASSWORD>,
        "nickname":         <NICKNAME>,             // Optional, default 'username'
        "rooms"*:           [ <ROOM>, ... ]
        }
    }
}
```

Note: * Items are mandatory (even if value is empty list/dict)

POST node details

Send node information to the server in order to check whether it can be provisioned.

POST /nodes

Request

```
Content-Type: application/json
{
    "model"*:              <MODEL_NAME>,
    "serialnumber"*:       <SERIAL_NUMBER>,
    "systemmac"*:          <SYSTEM_MAC>,
    "version"*:            <INTERNAL_VERSION>,

    "neighbors"*: {
        <INTERFACE_NAME(LOCAL)>: [ {
            'device':             <DEVICE_NAME>,
            'remote_interface':   <INTERFACE_NAME(REMOTE)>
        } ]
    },
}
```

Note: * Items are mandatory (even if value is empty list/dict)

Response

```
Status: 201 Created
Content-Type: text/html
Location: <url>

Status: 409 Conflict
Content-Type: text/html
Location: <url>

Status: 400 Bad Request
Content-Type: text/html
```

Status Codes

- 201 Created – Created
- 409 Conflict – Conflict
- 400 Bad Request – Bad Request

GET node definition

Request definition from the server.

GET /nodes/(*ID*)

Request

```
GET /nodes/{ID} HTTP/1.1
Host:
Accept: applicatino/json
Content-Type: text/html
```

Response

```
Status: 200 OK
Content-Type: application/json
{
    "name"*: <DEFINITION_NAME>

    "actions"*: [{ "action"*:          <NAME>*,
                   "description":     <DESCRIPTION>,
                   "onstart":         <MESSAGE>,
                   "onsuccess":       <MESSAGE>,
                   "onfailure":       <MESSAGE>,
                   "always_execute":  [True, False],
                   "attributes": { <KEY>: <VALUE>,
                                   <KEY>: { <KEY> : <VALUE>},
                                   <KEY>: [ <VALUE>, <VALUE> ]
                                   }
                   },...]
}
```

Note: * Items are mandatory (even if value is empty list/dict)

Status Codes

- 400 Bad Request – Bad Request
- 404 Not Found – Not Found

GET action

GET /actions/(*NAME*)

I Request action from the server.

Request

```
Content-Type: text/html
```

Response

```
Content-Type: text/x-python
```

statuscode 200 OK

statuscode 400 Bad Request

statuscode 404 Not Found

Status: 200 OK Content-Type: text/plain <PYTHON SCRIPT>

Status: 200 Bad request Content-Type: text/x-python

GET resource

GET /files/ (*RESOURCE_PATH*)
Request action from the server.

Request

```
Content-Type: text/html
```

Response

```
Status: 200 OK
Content-Type: text/plain
<resource>
```

Status Codes

- 200 OK – OK
- 404 Not Found – Not Found

2.7.3 Modules

ZTPServer Package

ztpserver Package

app Module

ztpserver.app.**enable_handler_console**(*level=None*)
Enables logging to stdout

ztpserver.app.**load_config**(*conf=None*)

ztpserver.app.**main**()
The main() is the main entry point for the ztpserver if called from the commmand line. When called from the command line, the server is running in standalone mode as opposed to using the application() to run under a python wsgi compliant server

ztpserver.app.**python_supported**()
Returns True if the current version of the python runtime is valid

ztpserver.app.**run_server**(*conf*, *debug*)
The run_server() is called by the main command line routine to run the server as standalone. This function accepts a single argument that points towards the configuration file describing this server

This function will block on the active thread until stopped.

Parameters **conf** – string path pointing to configuration file

ztpserver.app.**run_validator**(*filename=None*)

ztpserver.app.**start_logging**(*debug*)
reads the runtime config and starts logging if enabled

ztpserver.app.**start_wsgiapp**(*conf=None*, *debug=False*)
Provides the entry point into the application for wsgi compliant servers. Accepts a single keyword argument conf. The conf keyword argument specifies the path the server configuration file. The default value is /etc/ztpserver/ztpserver.conf.

Parameters **conf** – string path pointing to configuration file

Returns a wsgi application object

config Module

class ztpserver.config.**Attr**(*name*, ***kwargs*)
Bases: object

Base Attribute class for deriving all attributes for a Config object

Parameters

- **name** – required argument specifies attribute name
- **type** – optional keyword argument specifies attribute type. the default argument type is String
- **group** – optional keyword argument specifies attribute group. All attribute names must be unique within the group
- **default** – optional keyword argument specifies the default value for the attribute. The default value is None

class ztpserver.config.**BoolAttr**(*name*, ***kwargs*)
Bases: ztpserver.config.Attr

Boolean attribute class derived from Attr

class ztpserver.config.**Config**
Bases: _abcoll.Mapping

The Config class represents the configuration for collection.

add_attribute(*item*, *group=None*)

add_group(*group*)

clear_value(*name*, *group=None*)
clears the attributes value and resets it to default

read(*filename*)

set_value(*name*, *value*, *group=None*)

class ztpserver.config.**Group**(*name*, *config*)
Bases: _abcoll.Mapping

The Group class provides a logical grouping of attributes in a Config object. Group names must be unique for each Config instance and cannot be assigned values.

Parameters

- **name** – the name of the group
- **config** – the config object the group is associated with

add_attribute(*item*)

class ztpserver.config.**IntAttr**(*name*, *min_value=None*, *max_value=None*, ***kwargs*)
Bases: ztpserver.config.Attr

Integer attribute class derived from Attr

Parameters

- **min_value** – specifies the min value. the default is None
- **max_value** – specifies the max value. the default is None

class ztpserver.config.**ListAttr**(*name*, *delimiter=','*, ***kwargs*)
Bases: ztpserver.config.Attr

List attribute class derived from Attr

Parameters **delimiter** – specifies the delimiter character to split the string on

class ztpserver.config.**StrAttr**(*name*, *choices=None*, ***kwargs*)
Bases: ztpserver.config.Attr

String attribute class derived from Attr

Parameters **choices** – optional keyword argument specifies valid choices

controller Module

class ztpserver.controller.**ActionsController**(***kwargs*)
Bases: ztpserver.controller.BaseController

FOLDER = 'actions'

show(*request*, *resource*, ***kwargs*)
Handles GET /actions/{resource}

class ztpserver.controller.**BaseController**(***kwargs*)
Bases: ztpserver.wsgiapp.WSGIController

FOLDER = None

expand(**args*, ***kwargs*)
Returns an expanded file path relative to data_root

http_bad_request(**args*, ***kwargs*)
Returns HTTP 400 Bad Request

http_internal_server_error(**args*, ***kwargs*)
Returns HTTP 500 Internal server error

http_not_found(**args*, ***kwargs*)
Returns HTTP 404 Not Found

class ztpserver.controller.**BootstrapController**(***kwargs*)
Bases: ztpserver.controller.BaseController

DEFAULT_CONFIG = {'xmpp': {}, 'logging': []}

FOLDER = 'bootstrap'

config(*request*, ***kwargs*)
Handles GET /bootstrap/config

index(*request*, ***kwargs*)
Handles GET /bootstrap

class ztpserver.controller.**FilesController**(***kwargs*)
Bases: ztpserver.controller.BaseController

FOLDER = 'files'

show(*request*, *resource*, ***kwargs*)
Handles GET /files/{resource}

class ztpserver.controller.**MetaController**(***kwargs*)
Bases: ztpserver.controller.BaseController

BODY = {'sha1': None, 'size': None}

FOLDER = 'meta'

metadata(*request*, ***kwargs*)
Handles GET /meta/[actions|files|nodes]/<PATH_INFO>

class ztpserver.controller.**NodesController**(***kwargs*)
Bases: ztpserver.controller.BaseController

FOLDER = 'nodes'

create(*request*, ***kwargs*)
Handle the POST /nodes request

The create method will handle in incoming POST request from the node and determine if the node already exists or not. If the node does not exist, then the node will be created based on the request body.

Parameters **request** (*webob.Request*) – the request object from WSGI

Returns A dict as the result of the state machine which is used to create a WSGI response object.

do_actions(*response*, **args*, ***kwargs*)

do_resources(*response*, **args*, ***kwargs*)

do_substitution(*response*, **args*, ***kwargs*)

do_validation(*response*, **args*, ***kwargs*)

dump_node(*response*, **args*, ***kwargs*)
Writes the contents of the node to the repository

Parameters

- **response** (*dict*) – the response object being constructed
- **kwargs** (*dict*) – arbitrary keyword arguments

Returns a tuple of response object and next state. The next state is 'set_location'

finalize_response(*response*, **args*, ***kwargs*)

fsm(*state*, ***kwargs*)
Execute the FSM for the request

get_attributes(*response*, **args*, ***kwargs*)
Reads the resource specific attributes file and stores it in the response dict as 'attributes'

get_config(*request*, *resource*, ***kwargs*)

get_definition(*response*, **args*, ***kwargs*)
Reads the node specific definition from disk and stores it in the repsonse dict with key *definition*

get_startup_config(*response*, **args*, ***kwargs*)

node_exists(*response*, **args*, ***kwargs*)
Checks if the node already exists and determines the next state

This method will check for the existence of the node in the repository based on the node_id. The node_id keyword is pulled from the kwargs dict.

Parameters

- **response** (*dict*) – the response object being constructed
- **kwargs** (*dict*) – arbitrary keyword arguments

Returns A tuple that includes the updated response object and the next state to transition to. If the node already exists in the repository with a valid definition or startup-config, then the next state is 'dump_node' otherwise the next state is 'post_config'

post_config(*response*, **args*, ***kwargs*)
Writes the nodes startup config file if found in the request

Parameters

- **response** (*dict*) – the response object being constructed
- **kwargs** (*dict*) – arbitrary keyword arguments

Returns a tuple of response object and next state. If a config key was found in the request, the next state is 'set_location'. If not, the next state is 'post_node'.

post_node(*response*, **args*, ***kwargs*)
Checks topology validation matches and writes node specific files

This method will attempt to match the current node against the defined topology. If a match is found, then the pattern matched and definition (defined in the pattern) are written to the nodes folder in the repository and the response status is set to HTTP 201 Created.

Parameters

- **response** (*dict*) – the response object being constructed
- **kwargs** (*dict*) – arbitrary keyword arguments

Returns a tuple of response object and next state. The next state is 'dump_node'

Raises

- **If a match is not found, then a log message is created and** –

- **an IndexError is raised. If the node does not already** –
- **exist in the repository, then a log message is created and a** –
- **FileObjectNotFound exception is raised** –

put_config(*request*, ***kwargs*)

set_location(*response*, **args*, ***kwargs*)
Writes the HTTP Content-Location header

Parameters

- **response** (*dict*) – the response object being constructed
- **kwargs** (*dict*) – arbitrary keyword arguments

Returns a tuple of response object and next state. The next state is None.

Raises `Exception` – catches a general exception for logging and then re-raises it

show(*request*, *resource*, **args*, ***kwargs*)
Handle the GET /nodes/{resource} request

Parameters

- **request** (*webob.Request*) – the request object from WSGI
- **resource** (*str*) – the resource being requested

Returns A dict as the result of the state machine which is used to create a WSGI response object.

class `ztpserver.controller.`**Router**
Bases: `ztpserver.wsgiapp.WSGIRouter`

Routes incoming requests by mapping the URL to a controller

exception `ztpserver.controller.`**ValidationError**
Bases: `exceptions.Exception`

Base exception class for `Pattern`

`repository` Module

MODULE: ztpserver.respository

AUTHOR: Arista Networks

DESCRIPTION: The repository module provides read and write access to files for ztpserver. The repository module can perform basic file system like functionality for performing basid CRUD on files and well as reading and writing specific file contents.

copyright Copyright (c) 2014, Arista Networks

license BSD, see LICENSE for more details

class `ztpserver.repository.`**FileObject**(*name*, *path=None*, ***kwargs*)
Bases: `object`

The `FileObject` represents a single file entity in the repository. The instance provides convienent methods to read and write contents to the file using a specified serialization

hash()
Returns the SHA1 hash of the object.

Raises IOError

read(*content_type=None*, *node_id=None*)
Reads the contents from the file system

Parameters **content_type** (*str*) – defines the content_type of the file used to deserialize the object

Returns object

Raises FileObjectError

The read method will read the file from the file system, deserializing the contents as specified by the content_type argument. If the content_type argument is not specified, the read method will read the file as text. If any errors occur, a FileObjectError is raised.

size()
Returns the size of the object in bytes.

Raises IOError

write(*contents*, *content_type=None*)
Writes the contents to the file

Parameters

- **contents** (*str*) – specifies the contents to be written to the file
- **content_type** (*str*) – defines the serialization format to use when saving the file

Returns None

Raises FileObjectError

The write method takes the contents argument and writes it to the file using the serialization specified in the content_type argument. If the content_type argument is not specified, the contents are written as string text. This method will overwrite any contents that previously existed for the FileObj instance. If any errors are encountered during the write operation, a FileObjectError is raised

exception ztpserver.repository.**FileObjectError**
Bases: exceptions.Exception

Base exception class for FileObject

exception ztpserver.repository.**FileObjectNotFound**
Bases: ztpserver.repository.RepositoryError

Raised when a requested file is not found in the repository. This exception is a subclass of RespositoryError

class ztpserver.repository.**Repository**(*path*)
Bases: object

The Respository class represents a repository of FileObject instances. It is an abstract wrapper providing the ability to interact with persistently stored files.

add_file(*file_path*, *contents=None*, *content_type=None*)
Adds a new FileObject to the repository

Parameters

- **file_path** (*str*) – the full path of the file to add
- **contents** (*str*) – the contents to write to the file
- **content_type** (*str*) – specifies the serialization to use for the file

Returns FileObject

Raises RespositoryError

The add_file method allows for a new file to be added to the respository. If the file already exists, it is returned as an instance of `FileObject`. If the file doesn't already exist and the contents argument is not None, then the file is created and the contents written to the file. The content_type argument provides the serialization to be used when saving the file.

add_folder(*folder_path*)
Add a new folder to the repository

Parameters **folder_path** (*str*) – the full path of the folder to add

Returns str – the full path to the new folder

Raises RespositoryError

delete_file(*file_path*)
Deletes an existing file in the respository

Parameters **file_path** (*str*) – the file path of the instance to delete

Returns None

Raises RepositoryError

exists(*file_path*)
Returns boolean if the file_path exists in the repository

Parameters **file_path** (*str*) – the file_path to check for existence

Returns boolean – True if it exists otherwise False

expand(*file_path*)
Expands a file_path to the full path to a file object

Parameters **file_path** (*str*) – the file path to expand

Returns str – the full path to the file

This method is used to transform a relative file path into an absolute file path for identifying a file object resource

get_file(*file_path*)
Returns an intance of `FileObject` if it exists

Parameters **file_path** (*str*) – the file path of the instance to return

Returns instance of `FileObject`

Raises FileObjectNotFound

This method will retrieve a file object instance if it exists in the repository. If the file does not exist then an error is raised

exception ztpserver.repository.**RepositoryError**
Bases: exceptions.Exception

Base exception class for `Repository`

ztpserver.repository.**create_repository**(*path*)

resources Module

class ztpserver.resources.**ResourcePool**(*node_id*)
Bases: object

allocate(*pool*, *node*)

dump(*pool*)

load(*pool*)

lookup(*pool*, *node*)
Return an existing allocated resource if one exists

serialize()

exception ztpserver.resources.**ResourcePoolError**
Bases: exceptions.Exception

base error raised by Resource

serializers Module

class ztpserver.serializers.**BaseSerializer**(*node_id*)
Bases: object

Base serializer object

deserialize(*data*)
Deserialize an object to dict

serialize(*data*)
Serialize a dict to object

class ztpserver.serializers.**JSONSerializer**(*node_id*)
Bases: ztpserver.serializers.BaseSerializer

deserialize(*data*)
Deserialize a JSON object and return a dict

serialize(*data*)
Serialize a dict object and return JSON

class ztpserver.serializers.**Serializer**(*node_id*)
Bases: object

add_handler(*content_type*, *instance*)

deserialize(*data*, *content_type=None*)
Deserialize the data based on the content_type

handlers

serialize(*data*, *content_type*)
Serialize the data based on the content_type

exception ztpserver.serializers.**SerializerError**
Bases: exceptions.Exception

base error raised by serialization functions

class ztpserver.serializers.**TextSerializer**(*node_id*)
Bases: ztpserver.serializers.BaseSerializer

deserialize(*data*)
Deserialize a text object and return a dict

serialize(*data*)
Serialize a dict object and return text

class ztpserver.serializers.**YAMLSerializer**(*node_id*)
Bases: ztpserver.serializers.BaseSerializer

deserialize(*data*)
Deserialize a YAML object and return a dict

serialize(*data*)
Serialize a dict object and return YAML

ztpserver.serializers.**dump**(*data*, *file_path*, *content_type*, *node_id=None*)

ztpserver.serializers.**dumps**(*data*, *content_type*, *node_id*)

ztpserver.serializers.**load**(*file_path*, *content_type*, *node_id=None*)

ztpserver.serializers.**loads**(*data*, *content_type*, *node_id*)

topology Module

class ztpserver.topology.**ExactFunction**(*value*)
Bases: ztpserver.topology.Function

match(*arg*)

class ztpserver.topology.**ExcludesFunction**(*value*)
Bases: ztpserver.topology.Function

match(*arg*)

class ztpserver.topology.**Function**(*value*)
Bases: object

match(*arg*)

class ztpserver.topology.**IncludesFunction**(*value*)
Bases: ztpserver.topology.Function

match(*arg*)

class ztpserver.topology.**InterfacePattern**(*interface*, *remote_device*, *remote_interface*, *node_id*)
Bases: object

FUNCTIONS = {'regex': <class 'ztpserver.topology.RegexFunction'>, 'excludes': <class 'ztpserver.topology.ExcludesFunct

KEYWORDS = {'none': <ztpserver.topology.RegexFunction object at 0x7fec1f956710>, 'any': <ztpserver.topology.RegexFu

compile(*value*)

is_positive_constraint()

match(*interface*, *neighbors*)

match_interface(*interface*)

match_neighbor(*interface*, *neighbor*)

match_remote_device(*remote_device*)

match_remote_interface(*remote_interface*)

refresh()

exception ztpserver.topology.**InterfacePatternError**
Bases: exceptions.Exception

Base exception class for InterfacePattern

class ztpserver.topology.**Neighbor**
Bases: tuple

Neighbor(device, interface)

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

__getstate__()
Exclude the OrderedDict from pickling

__repr__()
Return a nicely formatted representation string

device
Alias for field number 0

interface
Alias for field number 1

class ztpserver.topology.**Neighbordb**(*node_id*)
Bases: object

RESERVED_VARIABLES = ['any', 'none']

add_pattern(*name*, ***kwargs*)

add_patterns(*patterns*)

add_variable(*key*, *value*, *overwrite=False*)

add_variables(*variables*)

find_patterns(*node*)

get_patterns()

static identifier(*node*)

is_global_pattern(*pattern*)

is_node_pattern(*pattern*)

match_node(*node*)

exception ztpserver.topology.**NeighbordbError**
Bases: exceptions.Exception

Base exception class for Neighbordb

class ztpserver.topology.**Node**(***kwargs*)
Bases: object

A Node object is maps the metadata from an EOS node. It provides access to the node's meta data including interfaces and the associated neighbors found on those interfaces.

add_neighbor(*interface*, *peers*)

add_neighbors(*neighbors*)

identifier()

serialize()

exception ztpserver.topology.**NodeError**
Bases: exceptions.Exception

Base exception class for Node

class ztpserver.topology.**OrderedCollection**(**args*, ***kwds*)
Bases: collections.OrderedDict

base object for using an ordered dictionary

class ztpserver.topology.**Pattern**(*name=None*, *definition=None*, *interfaces=None*, *node=None*, *variables=None*, *node_id=None*)
Bases: object

add_interface(*interface*)

add_interfaces(*interfaces*)

match_node(*node*)

parse_interface(*neighbor*)

serialize()

variable_substitution()

exception ztpserver.topology.**PatternError**
Bases: exceptions.Exception

Base exception class for Pattern

class ztpserver.topology.**RegexFunction**(*value*)
Bases: ztpserver.topology.Function

match(*arg*)

ztpserver.topology.**create_node**(*nodeattrs*)

ztpserver.topology.**default_filename**()
Returns the path for neighbordb based on the conf file

ztpserver.topology.**load_file**(*filename*, *content_type*, *node_id*)
Returns the contents of a file specified by filename.

The requred content_type argument is required and indicates the text serialization format the contents are stored in.

If the serializer load function encounters errors, None is returned

ztpserver.topology.**load_neighbordb**(*node_id*, *contents=None*)

ztpserver.topology.**load_pattern**(*pattern*, *content_type='application/yaml'*, *node_id=None*)
Returns an instance of Pattern

ztpserver.topology.**replace_config_action**(*resource*, *filename=None*)
Builds a definition with a single action replace_config

ztpserver.topology.**resources**(*attributes*, *node*, *node_id*)

types Module

class ztpserver.types.**Boolean**
Bases: object

FALSEVALUES = ['no', 'false', '0', 'off']

TRUEVALUES = ['yes', 'true', '1', 'on']

class ztpserver.types.**Integer**(*min_value=None*, *max_value=None*)
Bases: object

class ztpserver.types.**List**(*delimiter=', '*)
Bases: object

class ztpserver.types.**String**(*choices=None*)
Bases: object

utils Module

ztpserver.utils.**atoi**(*text*)

ztpserver.utils.**expand_range**(*interfaces*)
Returns a naturally sorted list of items expanded from interfaces.

ztpserver.utils.**natural_keys**(*text*)

ztpserver.utils.**parse_interface**(*neighbor*, *node_id*)

validators Module

class ztpserver.validators.**InterfacePatternValidator**(*node_id*)
Bases: ztpserver.validators.Validator

validate_interface_pattern()

class ztpserver.validators.**NeighbordbValidator**(*node_id*)
Bases: ztpserver.validators.Validator

validate_patterns()

validate_variables()

class ztpserver.validators.**PatternValidator**(*node_id*)
Bases: ztpserver.validators.Validator

validate_attributes()

validate_definition()

validate_interfaces()

validate_name()

validate_node()

validate_variables()

exception ztpserver.validators.**ValidationError**
Bases: exceptions.Exception

Base error class for validation failures

class ztpserver.validators.**Validator**(*node_id*)
Bases: object

error(*err*, **args*, ***kwargs*)

validate(*data=None*)

ztpserver.validators.**validate_neighbordb**(*contents*, *node_id*)

ztpserver.validators.**validate_pattern**(*contents*, *node_id*)

wsgiapp Module

class ztpserver.wsgiapp.**WSGIController**
Bases: object

create(*request*, ***kwargs*)

delete(*request*, *resource*, ***kwargs*)

edit(*request*, *resource*, ***kwargs*)

index(*request*, ***kwargs*)

new(*request*, ***kwargs*)

response(***kwargs*)

show(*request*, *resource*, ***kwargs*)

update(*request*, *resource*, ***kwargs*)

class ztpserver.wsgiapp.**WSGIRouter**(*mapper*)
Bases: object

route
Turns a request-taking, response-returning function into a WSGI app

You can use this like:

```
@wsgify
def myfunc(req):
    return webob.Response('hey there')
```

With that `myfunc` will be a WSGI application, callable like `app_iter = myfunc(environ, start_response)`. You can also call it like normal, e.g., `resp = myfunc(req)`. (You can also wrap methods, like `def myfunc(self, req)`.)

If you raise exceptions from `webob.exc` they will be turned into WSGI responses.

There are also several parameters you can use to customize the decorator. Most notably, you can use a `webob.Request` subclass, like:

```
class MyRequest(webob.Request):
    @property
    def is_local(self):
        return self.remote_addr == '127.0.0.1'
@wsgify(RequestClass=MyRequest)
def myfunc(req):
    if req.is_local:
        return Response('hi!')
    else:
        raise webob.exc.HTTPForbidden
```

Another customization you can add is to add *args* (positional arguments) or *kwargs* (of course, keyword arguments). While generally not that useful, you can use this to create multiple WSGI apps from one function, like:

```
import simplejson
def serve_json(req, json_obj):
    return Response(json.dumps(json_obj),
```

```
                                content_type='application/json')

serve_ob1 = wsgify(serve_json, args=(ob1,))
serve_ob2 = wsgify(serve_json, args=(ob2,))
```

You can return several things from a function:

- A `webob.Response` object (or subclass)
- *Any* WSGI application
- None, and then `req.response` will be used (a pre-instantiated Response object)
- A string, which will be written to `req.response` and then that response will be used.
- Raise an exception from `webob.exc`

Also see `wsgify.middleware()` for a way to make middleware.

You can also subclass this decorator; the most useful things to do in a subclass would be to change *RequestClass* or override *call_func* (e.g., to add `req.urlvars` as keyword arguments to the function).

Bootstrap Client

class `ztps.bootstrap.`**Node**(*server*)

Node object which can be used by actions via: attributes.get('NODE')

client
jsonrpclib.Server

jsonrpclib connect to Command API engine

api_config_cmds(*cmds*)
Run CLI commands via Command API, starting from config mode.

Commands are ran in order.

Parameters **cmds** (*list*) – List of CLI commands.

Returns List of Command API results corresponding to the input commands.

Return type list

api_enable_cmds(*cmds*, *text_format=False*)
Run CLI commands via Command API, starting from enable mode.

Commands are ran in order.

Parameters

- **cmds** (*list*) – List of CLI commands.
- **text_format** (*bool, optional*) – If true, Command API request will run in text mode (instead of JSON).

Returns List of Command API results corresponding to the input commands.

Return type list

append_rc_eos_lines(*lines*)
Add lines to rc.eos.

Parameters **lines** (*list*) – List of bash commands

append_startup_config_lines(*lines*)
Add lines to startup-config.

Parameters **lines** (*list*) – List of CLI commands

details()
Get details.

Returns

System details

Format:

```
{'model':        <MODEL>,
 'version':      <EOS_VERSION>,
 'systemmac':    <SYSTEM_MAC>,
 'serialnumber': <SERIAL_NUMBER>,
 'neighbors':    <NEIGHBORS>        # see neighbors()
}
```

Return type dict

flash()
Get flash path.

Returns flash path

Return type string

has_startup_config()
Check whether startup-config is configured or not.

Returns True is startup-config is configured; false otherwise.

Return type bool

log_msg(*msg*, *error=False*)
Log message via configured syslog/XMPP.

Parameters

- **msg** (*string*) – Message
- **error** (*bool, optional*) – True if msg is an error; false otherwise.

neighbors()
Get neighbors.

Returns

LLDP neighbor

Format:

```
{'neighbors': {<LOCAL_PORT>:
 [{'device': <REMOTE_DEVICE>,
   'port': <REMOTE_PORT>}, ...],
...}}
```

Return type dict

rc_eos()
Get rc.eos path.

Returns rc.eos path

Return type string

retrieve_url(*url*, *path*)
Download resource from server.

If 'path' is somewhere on flash, the client will first request the metainformation for the resource from the server (in order to Check whether there is enogh disk space available).

Raises ZtpError – resource cannot be retrieved: - metainformation cannot be retrieved from server OR - disk space on flash is insufficient OR - file cannot be written to disk

Returns startup-config path

Return type string

classmethod server_address()
Get ZTP Server URL.

Returns ZTP Server URL.

Return type string

startup_config()
Get startup-config path.

Returns startup-config path

Return type string

system()
Get system information.

Returns

System information

Format:

```
{'model':        <MODEL>,
 'version':      <EOS_VERSION>,
 'systemmac':    <SYSTEM_MAC>,
 'serialnumber': <SERIAL_NUMBER>}
```

Return type dict

Actions

- add_config Action
- copy_file Action
- install_cli_plugin Action
- install_extension Action
- install_image Action
- replace_config Action
- send_email Action

`add_config` Action

ztps.add_config.**main**(*attributes*)

Adds startup-config section.

Appends config section to startup config based on the value of the 'url' attribute.

This action is dual-supervisor compatible.

Parameters **attributes** – list of attributes; use attributes.get(<ATTRIBUTE_NAME>) to read attribute values

:param Special attributes::

node: attributes.get('NODE') API: see documentation

`copy_file` Action

ztps.copy_file.**main**(*attributes*)

Copies file to the switch.

Copies file based on the values of 'src_url' and 'dst_url' attributes ('dst_url' should point to the detination folder). If 'overwrite' is set to:

- 'replace': the file is copied to the switch regardless of whether there is already a file with the same name at the destination;
- 'if-missing': the file is copied to the switch only if there is not already a file with the same name at the destination; if there is, then the action is a no-op;
- 'backup': the file is copied to the switch; if there is already another file at the destination, that file is renamed by appending the '.backup' suffix

If 'overwrite' is not set, then 'replace' is the default behaviour.

This action is NOT dual-supervisor compatible.

Parameters **attributes** – list of attributes; use attributes.get(<ATTRIBUTE_NAME>) to read attribute values

:param Special attributes::

node: attributes.get('NODE') API: see documentation

`install_cli_plugin` Action

ztps.install_cli_plugin.**main**(*attributes*)

Installs EOS CliPlugin.

Installs CliPlugin based on the value of the 'url' attribute.

This action is NOT dual-supervisor compatible.

Parameters **attributes** – list of attributes; use attributes.get(<ATTRIBUTE_NAME>) to read attribute values

:param Special attributes::

node: attributes.get('NODE') API: see documentation

install_extension Action

ztps.install_extension.**main**(*attributes*)
Installs EOS extension.

Installs extension based on the value of the 'url' attribute. If 'force' is set, then the dependency checks are overridden.

This action is NOT dual-supervisor compatible.

Parameters **attributes** – list of attributes; use attributes.get(<ATTRIBUTE_NAME>) to read attribute values

:param Special attributes::

node: attributes.get('NODE') API: see documentation

install_image Action

ztps.install_image.**main**(*attributes*)
Installs software image on the switch.

If the current software image is the same as the 'version' attribute value, then this action is a no-op. Otherwise, the action will replace the existing software image.

For dual supervisor systems, the image on the active supervisor is used as reference.

This action is dual-supervisor compatible.

Parameters **attributes** – list of attributes; use attributes.get(<ATTRIBUTE_NAME>) to read attribute values

:param Special attributes::

node: attributes.get('NODE') API: see documentation

replace_config Action

ztps.replace_config.**main**(*attributes*)
Replaces stratup-config on the switch.

Replaces/adds /mnt/flash/startup-config based on the value of the 'url' attribute.

This action is dual-supervisor compatible.

Parameters **attributes** – list of attributes; use attributes.get(<ATTRIBUTE_NAME>) to read attribute values

:param Special attributes::

node: attributes.get('NODE') API: see documentation

send_email Action

ztps.send_email.**main**(*attributes*)
Sends an email using an SMTP relay host

Generates an email from the bootstrap process and routes it through a smarthost. The parameters value expects a dictionary with the following values in order for this function to work properly.

```
{
    'smarthost':   <hostname of smarthost>,
    'sender':      <from email address>
    'receivers':   [ <array of recipients to send email to> ],
    'subject':     <subject line of the message>,
    'body':        <the message body>,
    'attachments': [ <array of files to attach> ],
    'commands':    [ <array of commands to run and attach> ]
}
```

The required fields for this function are smarthost, sender, and receivers. All other fields are optional.

This action is dual-supervisor compatible.

Parameters **attributes** – list of attributes; use attributes.get(<ATTRIBUTE_NAME>) to read attribute values

:param Special attributes::

node: attributes.get('NODE') API: see documentation

2.8 Glossary of Terms

action an action is a Python script which is executed during the bootstrap process.

attribute an attribute is a variable that holds a value. attributes are used in order to customise the behaviour of actions which are executed during the bootstrap process.

definition a definition is a YAML file that contains a collection of all actions (and associated attributes) which need to run during the bootstrap process in order to fully provision a node

neighbordb neighbordb is a YAML file which contains a collection of patterns which can be used in order to map nodes to definitions

node a node is a EOS instance which is provisioned via ZTPServer. A node is uniquely identified by its unique_id (serial number or system MAC address) and/or unique position in the network.

pattern a pattern is a YAML file which describes a node in terms of its unique_id (serial number or system MAC) and/or location in the network (neighbors)

resource pool a resource pool is a YAML file which provides a mapping between a set or resources and the nodes to which some of the resources might have been allocated to. The nodes are uniquely identified via their system MAC.

unique_id the unique identifier for a node. This can be configured, globally, to be the serial number (default) or system MAC address in the ztpserver.conf file

2.9 Resources

ZTPServer documentation and other reference materials are below:

- GitHub ZTPServer Repository
- ZTPServer wiki

- Packer VM build process.
- ZTPServer Python (PyPI) package.
- YAML Code Validator

2.10 Support

ZTPServer is an Arista led open source community project. Users and developers are encouraged to contribute to the project. See CONTRIBUTING.md in the source for more information on how.

Community based support is available through

- eosplus forum
- eosplus-dev@arista.com.
- IRC: irc.freenode.net#arista

Commercial support, customization, and integration services are available through the EOS+ team at Arista Networks, Inc.. Contact eosplus-dev@arista.com for details.

2.11 Release Notes

The authoritative state for any known issue can be found in GitHub issues.

2.11.1 Release 1.1

(Published August, 2014)

The authoritative state for any known issue can be found in GitHub issues.

Enhancements

- **V1.1.0 docs (181)** Documentation has been completely restructured and is now hosted at http://ztpserver.readthedocs.org/.
- **refresh_ztps - util script to refresh ZTP Server installation (177)** /utils/refresh_ztps can be used in order to automatically refresh the installation of ZTP Server to the latest code on GitHub. This can be useful in order to pull bug fixes or run the latest version of various development branches.
- **Et49 does not match Ethernet49 in neighbordb/pattern files (172)** The local interface in an interface pattern does not have to use the long interface name. For example, all of the following will be treated similarly: Et1, e1, et1, eth1, Eth1, ethernet1, Ethernet1.

  Note that this does not apply to the remote interface, where different rules apply.
- **Improve server-side log messages when there is no match for a node on the server (171)**
- **Improve error message on server side when definition is missing from the definitions folder (170)**
- **neighbordb should also support serialnumber as node ID (along with system MAC) (167)** Server now supports two types of unique identifiers, as specified in ztpserver.conf:

  ```
  # UID used in the /nodes structure (either systemmac or serialnumber)
  identifier = serialnumber
  ```

The configuration is global and applies to a single run of the server (neighbordb, resource files, nodes' folders, etc.).

- **serialnumber should be the default identifier instead of systemmac (166)** The default identifier in ztpserver.conf is the serial number. e.g.

  ```
  # UID used in the /nodes structure (either systemmac or serialnumber)
  identifier = serialnumber
  ```

  This is different from v1.0, where the systemmac was the default.

- **Document which actions are dual-sup compatible and which are not (165)** All actions now document whether they are dual-sup compatible or not. See documentation for the details.
- **dual-sup support for install_image action (164)** install_image is now compatible with dual-sup systems.
- **Resource pool allocation should use the identifier instead of the systemmac (162)** The values in the resource files will be treated as either system MACs or serial numbers, depending on what identifier is configured in the global configuration file.
- **Document actions APIs (157)** The API which can be used by actions is now documented in the documentation for the bootstrap script module.
- **Get rid of return codes in bootstrap script (155)**
- **Bootstrap script should always log a detailed message before exiting (153)** bootstrap script will log the reason for exiting, instead of an error code.
- **Client should report what the error code means (150)**
- **Improve server logs when server does not know about the node (145)**
- **Configurable verbosity for logging options (server side) (140)** Bootstrap configuration file can now specify the verbosity of client-side logs:

  ```
  ...
  xmpp:
  username: ztps
  password: ztps
  domain: pcknapweed.lab.local
  <b>msg_type : debug</b>
  rooms:
      - ztps-room
  ```

  The allowed values are:

  – debug: verbose logging

  – info: log only messages coming from the server (configured in definitions)

  The information is transmitted to the client via the bootstrap configuration request:

  ```
  ####GET logging configuration
  Returns the logging configuration from the server.

      GET /bootstrap/config

  Request

      Content-Type: text/html

  Response

      Status: 200 OK
  ```

```
    Content-Type: application/json
    {
        "logging"*: [ {
            "destination": "file:/<PATH>" | "<HOSTNAME OR IP>:<PORT>",   //localhost enabled
                                                                         //by default
            "level"*:        <DEBUG | CRITICAL | ...>,
        } ]
    },
        "xmpp"*:{
            "server":           <IP or HOSTNAME>,
            "port":             <PORT>,                 // Optional, default 5222
            "username"*:        <USERNAME>,
            "domain"*:          <DOMAIN>,
            "password"*:        <PASSWORD>,
            "nickname":         <NICKNAME>,             // REMOVED
            "rooms"*:           [ <ROOM>, ... ]
            "msg_type":         [ "info" | "debug" ]    // Optional, default "debug"

        }
    }

>**Note**: * Items are mandatory (even if value is empty list/dict)
```

P.S. (slightly unrelated) The nickname configuration has been deprecated (serialnumber is used instead).

- **Configurable logging levels for xmpp (139)** bootstrap.conf:

  ```
  logging:
  ...
  xmpp:
  ...
  nickname: ztps        // (unrelated) this was removed - using serial number instead
  msg_type: info        // allowed values ['info', 'debug']
  ```

  If msg_type is set to 'info', only log via XMPP error messages and 'onstart', 'onsuccess' and 'onfailure' error messages (as configured in the definition).

- **Bootstrap should rename LLDP SysDescr to "provisioning" while executing or failing (138)**
- **Test XMPP for multiple nodes being provisioned at the same time (134)**
- **Server logs should include ID (MAC/serial number) of node being provisioned (133)** Most of the server logs will not be prefixed by the identifier of the node which is being provisioned - this should make debugging environments where multiple nodes are provisioned at the same time a lot easier.
- **Use serial number instead of system MAC as the unique system ID (131)**
- **Bootstrap script should disable copp (122)**
- **Bootstrap script should check disk space before downloading any resources (118)** Bootstrap script will request the meta information from server, whenever it attempts to save a file to flash. This information will be used in order to check whether enough disk space is available for downloading the resource.

  ```
  ####GET action metadata
  Request action from the server.

      GET /meta/actions/NAME

  Request

      Content-Type: text/html
  ```

```
Response

Status: 200 OK
    Content-Type: application/json
    {
        "size"*:  <SIZE IN BYTES>,
        "sha1": <HASH STRING>
    }

>**Note**: * Items are mandatory (even if value is empty list/dict)

    Status: 404 Not found
    Content-Type: text/html

    Status: 500 Internal server error                    // e.g. permissions issues on ser
    Content-Type: text/html
```

- **ztps should check Python version and report a sane error is incompatible version is being used to run it (110)** ztps reports error if it is ran on a system with an incompatible Python version installed.
- **Do not hardcode Python path (109)**
- **Set XMPP nickname to serial number (106)** Serial number is used as XMPP presence/nickname. For vEOS instances which don't have one configured, systemmac is used instead.
- **Send serial number as XMPP presence (105)** Serial number is used as XMPP presence/nickname. For vEOS instances which don't have one configured, systemmac is used instead.
- **Support for EOS versions < 4.13.3 (104)** ZTP Server bootstrap script now supports any EOS v4.12.x or later.
- **neighbordb should not be cached (97)** Neighbordb is not cached on the server side. This means that any updates to it do not require a server restart anymore.
- **Definitions/actions should be loaded form disk on each GET request (87)** Definitions and actions are not cached on the server side. This means that any updates to them do not require a server restart anymore.
- **Fix all pylint warnings (83)**
- **add_config action should also accept server-root-relative path for the URL (71)** 'url' atrribute in add_config action can be either a URL or a local server path.
- **install_image action should also accept server-root-relative path for the URL (70)** 'url' atrribute in install_image action can be either a URL or a local server path.
- **Server logs should be timestamped (63)** All server-side logs now contain a timestamp. Use 'ztps –debug' for verbose debug output.
- **After installing ZTPServer, there should be a dummy neighbordb (with comments and examples) and a dummy resource (**
- **need test coverage for InterfacePattern (42)**
- **test_topology must cover all cases (40)**

Resolved issues

- **Syslog messages are missing system-id (vEOS) (184)** All client-side log message are prefixed by the serial number for now (regardless of what the identifier is configured on the server).

  For vEOS, if the system does not have a serial number configured, the system MAC will be used instead.

- **No logs while executing actions (182)**
- **test_repository.py is leaking files (174)**
- **Allocate function will return some SysMac in quotes, others not (137)**
- **Actions which don't require any attributes are not supported (129)**
- **Static pattern validation fails in latest develop branch (128)**
- **Have a way to disable topology validation for a node with no LLDP neighbors (127)** COPP is disabled during the bootstrap process for EOS v4.13.x and later. COPP is not supported for older releases.
- **Investigate "No loggers could be found for logger sleekxmpp.xmlstream.xmlstream" error messages on client side (120)**
- **ZTPS should not fail if no variables are defined in neighbordb (114)**
- **ZTPS should not fail if neighbordb is missing (113)**
- **ZTPS installation should create dummy neighbordb (112)** ZTP Server install will create a placeholder neighbordb with instructions.
- **Deal more gracefully with invalid YAML syntax in resource files (75)**
- **Server reports AttributeError if definition is not valid YAML (74)**
- **fix issue with Pattern creation from neighbordb (44)**

2.12 Known Caveats

The authoritative state for any known issue can be found in GitHub issues.

- Currently, the management interfaces may not be used as valid local interface names in neighbordb. When creating patterns, use `any` instead.
- Only a single entry in a resource pool may be allocated to a node.
- Be sure your host firewall allows incoming connections to ZTPServer. The standalone server runs on port TCP/8080 by default. For **firewalld**:
  - Open TCP/<port> through firewalld `bash-3.2$ firewall-cmd --zone=public --add-port=<port>/tcp [--permanent]`
  - Stop firewalld `bash-3.2$ systemctl status firewalld`
  - Disable firewalld `bash-3.2$ systemctl disable firewalld`

2.13 Roadmap

The authoritative state, including the intended release, for any known issue can be found in GitHub issues. The information provided here is current at the time of publishing but is subject to change. Please refer to the latest information in GitHub issues by filtering on the desired milestone.

2.13.1 Release 1.2

Target: November 2014

Enhancements

- Add extra logging to "copy_file" action (187)
- Local interface in pattern specification should also allow ManagementXXX (185)
- Bootstrap script should cleanup on failure (176)
- utils.py: expand_range needs to be improved (173)
- Allow posting the startup-config to a node's folder, even if no startup-config is present (169)
- test_controller.py should cover both serialnumber and systemmac as identifiers (168)
- Tests for resource pool allocation (161)
- Bootstrap XMPP logging - client fails to create the specified MUC room (148)
- add_mlag_config action (action to arbitrate between MLAG peers) (141)
- ZTPS server fails to write .node because lack of permissions (126)
- Location-based ZTR (103)
- Remove definition line from auto-generated pattern (102)
- tests for attribute tiebreakers for all 4 types of attributes (101)
- Breadth tests for resource pools (94)
- We need visibility into which files the server is loading and when (52)

2.13.2 Release 1.3

Target: January 2015

Enhancements

- All requests from the client should contain the unique identifier of the node (188)
- make install_extension action dual-sup compatible (180)
- make install_cli_plugin action dual-sup compatible (179)
- make copy_file action dual-sup compatible (178)
- New command line option for ZTP Server which clears the resource pool allocations (163)
- Hook to run script after posting files on the server (132)
- Plugin infrastructure for resource pool allocation (121)
- Enhance actions to check md5sum of downloaded resources (107)
- V1 (181)

2.14 License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

Neither the name of the {organization} nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

2.14.1 Licenses of Bundled Third Party Code

Requests v2.3.0: HTTP for Humans

Copyright 2014 Kenneth Reitz

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

/actions

/bootstrap

/files

/nodes

Z

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Y

Z