Warthog Documentation

Release 2.0.1

Smarter Travel

Jul 20, 2017

Contents

Warthog is a simple Python client for interacting with A10 load balancers. The target use case is for safely removing servers from pools in a load balancer during a deployment. It is available under the MIT license.

CHAPTER 1

Features

- Waiting for servers to finish all requests when being disabled
- Graceful handling of transient errors with optional retry logic
- Support for interacting with the load balancer API using SSL
- Works with Python 2.6, 2.7, 3.3, 3.4, 3.5, 3.6+
- Thread safety

CHAPTER 2

Installation

To install Warthog, simply run:

```
$ pip install warthog
```

CHAPTER 3

Dependencies

- requests by Kenneth Reitz, version 2.6 or higher
- click by Armin Ronacher, version 3.3 or higher

CHAPTER 4

Usage

Using the client is easy!

```
from warthog.api import WarthogClient

def install_my_project(server):
    pass

client = WarthogClient('https://lb.example.com', 'user', 'password')

client.disable_server('app1.example.com')
install_my_project('app1.example.com')
client.enable_server('app1.example.com')
```

CHAPTER 5

Contents

Install

The sections below will go over how to install the Warthog client from the Python Package Index (PyPI) or how to install it from source. Both ways to install the client can be done globally on a machine or within the context of a virtualenv.

Prerequisites

These instructions assume that you have the following things available.

- Python 2.6 - 2.7 or Python 3.3 - Python 3.6
- The virtualenv tool
- The pip tool
- Git

Requirements

Warthog depends on the following libraries / projects. If you have Python installed and use the pip tool for installation, these should be installed automatically.

- Python 2.6 - 2.7 or Python 3.3 - Python 3.6
- The Requests library (HTTP library), version 2.6 or higher
- The Click library (command line interface library), version 3.3 or higher

Install from PyPI

If you're planning on installing from PyPI, the process is pretty easy. Depending on how you want to use the client, you have a few options.

Globally

You can install the client globally on a machine (using the system Python version) and it will be available for all users of the machine. This will typically require root or administrator permissions.

```
$ pip install warthog
```

Virtual Environment

You can also install the client into a virtual environment. A virtual environment is a self-contained Python installation into which you can install whatever you'd like without root or administrator permissions. Packages installed in this environment will only be available in the context of that environment.

```
# Create your new virtual environment
$ virtualenv my-warthog-install

# Enter the virtual environment
$ source my-warthog-install/bin/activate

# Install the Warthog client
$ pip install warthog
```

Install from Source

First you'll need to get the source code of the client using Git.

```
$ git clone https://github.com/smarter-travel-media/warthog.git
```

Globally

Like installation from PyPI, installation from source can be done globally for all users of a machine. As above, this will typically require root or administrator permissions.

```
$ cd warthog && pip install .
```

Virtual Environment

You can also install the client from source into a virtual environment.

```
# Create your new virtual environment
$ virtualenv my-warthog-install

# Enter the virtual environment
$ source my-warthog-install/bin/activate
```

```
# Install the client from the source checkout we made above
$ cd warthog && pip install .
```

Design

Some of the principals, design decisions, and limitations of the Warthog library are outlined below.

Purpose

The target use case of the Warthog client is safely disabling servers while each server is restarted or deployed to. Because of this, Warthog has a very slim feature set. The client only exposes a small subset of operations available through the HTTP API for A10 load balancers.

Node Level vs Group Level

When interacting with a server in a load balancer, there are typically two ways the server can be manipulated – at the node level and at the group level. Interacting with a server at the node level means that it will be disabled for all groups that the server belongs to. Interacting with a server at the group level means that the server may be active for one group and disabled for others.

The Warthog library embraces the idea that nodes should be single purpose and not run multiple unrelated services. Because of this, Warthog supports interacting with servers only at the node level, not the group level. When the status of servers is queried, when servers are disabled, and when servers are enabled, all operations are at the node level.

Thread Safety

The main interfaces of the Warthog library (*`warthog.client.WarthogClient`*, *`warthog.client.CommandFactory`*, and assorted classes in *`warthog.config`*) are thread safe. Each class will also include a comment in the doc string that indicates if it is thread safe. The Warthog library makes use of Requests (for making HTTP and HTTPS calls to the load balancer API) which is also thread safe.

SSL

The Warthog library supports interacting with a load balancer over HTTPS. By default when using SSL, certificates will be verified, and TSLv1 will be explicitly used. If you wish to override either of these, you can do so when using the *`warthog.client.WarthogClient`* or in your configuration file.

Versions

The Warthog library uses semantic versioning of the form `major.minor.patch`. All backwards incompatible changes after version `1.0.0` will increment the major version number. All backwards incompatible changes prior to version `1.0.0` will increment the minor version number.

Since this is a Python project, only the subset of the semantic versioning spec that is compatible with PEP-440 will be used.

Compatible and incompatible changes are defined in terms of use of the Python module `warthog.api` and use of the CLI program `warthog`. Meaning, code that makes use of the 1.0.0 version of `warthog.api` should not require any changes to make use of the 1.1.0 version of `warthog.api`.

Alternatives

If your use case doesn't fit the design or goals of the Warthog client, don't despair, you have options!

The official client supports many more operations, though, at the price of a reduced level of abstraction.

You can also write your own client based on available documentation.

Usage

Descriptions and examples of how to make use of the Warthog library in common deploy-related situations are outlined below.

Each of the examples will create a new instance of the Warthog client but, obviously, you do not need to do this in your actual deploy process. In fact, you can even share the same client instance between multiple threads (it's thread safe and immutable).

Create a Client

Let's start with the simplest possible thing you can do, just create a client instance.

```python
from warthog.api import WarthogClient

client = WarthogClient('https://lb.example.com', 'deploy', 'my password')
```

In the code above, we create a new client instance with the following properties:

- It will connect to the load balancer at `lb.example.com`.
- It will connect over HTTPS and it will validate the certificate presented by the load balancer.
- It will use the user 'deploy' and the password 'my password' for authentication.

You'll notice that we didn't tell the client to validate the certificate presented by the load balancer but it will attempt to do that anyway. That is because the client attempts to be safe by default. We'll explain how to disable this later.

Create a Client Without SSL Verification

It might be the case that you use a self-signed certificate for HTTPS connections to your load balancer. This isn't ideal, but hey, it happens. To this end, you can configure the Warthog library to connect over HTTPS but *not* to verify the SSL certificate.

This can be easily accomplished by passing an extra argument to the client when creating it.

```python
from warthog.api import WarthogClient

client = WarthogClient(
    'https://lb.example.com', 'deploy', 'my password', verify=False)
```

Create a Client With an Alternate SSL/TLS Version

If you need to use an alternate version of SSL or TLS to interact with your load balancer over HTTPS, you can accomplish this by just passing an extra argument to the client when creating it.

```
import ssl
from warthog.api import WarthogClient

client = WarthogClient(
    'https://lb.example.com', 'deploy', 'my password', ssl_version=ssl.PROTOCOL_TLSv1_
↪2)
```

Create a Client From a Configuration File

Hopefully, the thought of sprinkling your load balancer credentials throughout your deploy scripts makes you nervous. Luckily, there's functionality in the Warthog library for reading configuration information about the load balancer from an external file.

For more details about the format of this file and the default expected locations, see the *CLI Tool* section.

Let's start by importing all the parts of the API that we'll need.

```
from warthog.api import WarthogClient, WarthogConfigLoader
```

The `WarthogConfigLoader` class can load configuration from an explicit path or it can check several expected locations for the file. Let's start with loading an explicit path.

```
config_loader = WarthogConfigLoader(config_file='/etc/load-balancer.ini')
```

At this point, we haven't actually loaded anything. Let's do that next.

```
config_loader.initialize()
config_settings = config_loader.get_settings()
```

Now we're talking! At this point, we have an immutable struct-like object (a named tuple in Python parlance) with all the needed values for creating a new `WarthogClient` instance. Let's do that now.

```
client = WarthogClient(
    config_settings.scheme_host, config_settings.username, config_settings.password,
    verify=config_settings.verify, ssl_version=config_settings.ssl_version)
```

Disable a Server

If you're using the Warthog library as part of your deploy process, one of the first things you'll need to do is safely remove a server from receiving traffic in the load balancer. Let's explore that below.

First, create the client instance that we'll be using.

```
from warthog.api import WarthogClient

client = WarthogClient('https://lb.example.com', 'deploy', 'my password')
```

Next, we'll mark a server as disabled in the load balancer, letting the client use retry logic to attempt to do this as safely as possible. Note that the server is specified by hostname alone.

```
client.disable_server('app1.example.com')
```

You might notice that this method doesn't return immediately, it takes a little bit. That's because when we disable a server by default we:

- Mark the server as disabled, attempting this a few times if there are errors making the disable request.
- Check the number of active connections to the server every few seconds, waiting until this number reaches zero.
- After waiting up to a maximum amount of time for the number of connections on the server to reach zero, check if the server actually got disabled.

It might be the case that you don't really need to wait for the number of connections to reach zero. If this is the case, you can tell the client not to use retry logic or wait for the number of connections to drop to zero.

```
client.disable_server('app1.example.com', max_retries=0)
```

You can set `max_retries` to any number that makes sense for your deploy process. Each retry will be attempted two seconds apart by default. See *warthog.client.WarthogClient* for more information about how to change the time between retries.

Enable a Server

After you've deployed a new version of your application to a server or restarted it, you'll need to enable the server so that it starts receiving traffic from the load balancer. The method for doing this is very similar to how disabling a server works. We'll go into it more below.

First, create the client instance that we'll be using.

```
from warthog.api import WarthogClient

client = WarthogClient('https://lb.example.com', 'deploy', 'my password')
```

Next, we'll mark a server as enabled in the load balancer, letting the client use retry logic to make sure that the server actually ends up enabled. Note that the server is specified by hostname alone.

```
client.enable_server('app1.example.com')
```

Similar to disabling a server, this method won't return immediately. When we enable server by default we:

- Mark the server as enabled, attempting this a few times if there are errors making the enable request.
- Check the status of the server, waiting until it becomes 'enabled'
- After waiting up to a maximum amount of time for the server to become enabled, check if the server actually got enabled.

Similar to disabling a server, it might be the case that you don't really need to wait for a server to become enabled. If this is the case, you can tell the client not to use retry logic or wait for the server to become enabled.

```
client.enable_server('app1.example.com', max_retries=0)
```

You can set `max_retries` to any number that makes sense for your deploy process. Each retry will be attempted two seconds apart by default. See *warthog.client.WarthogClient* for more information about how to change the time between retries.

Non-Load Balanced Servers

If you use the same deployment process for servers that are in a load balancer and servers that aren't in a load balancer, you'll have to deal with that when you use the Warthog library.

When you attempt to enable, disable, or otherwise interact with a non-load balanced host through the load balancer you'll get an exception (`warthog.exceptions.WarthogNoSuchNodeError`) indicating that this is not a host that the load balancer knows about. Let's look at how to handle this situation below.

First, create the client instance that we'll be using.

```
from warthog.api import WarthogClient, WarthogNoSuchNodeError

client = WarthogClient('https://lb.example.com', 'deploy', 'my password')
```

Next we'll attempt to disable the server as part of our deploy process, but we'll catch the exception raised when the server isn't recognized by the load balancer.

```
try:
    client.disable_server('app1.example.com')
except WarthogNoSuchNodeError:
    use_lb = False
else:
    use_lb = True

# Your deploy process goes here...

if use_lb:
    client.enable_server('app1.example.com')
```

You can see above that we catch the exception that indicates this is not a host that the load balancer knows about. In this case, we make sure to not attempt to enable the server after completing our deployment (or application restart, etc.).

Already Disabled Servers

Sometimes a server gets marked as disabled in a load balancer outside of your deploy process. Maybe the server is being used for load testing, maybe some maintenance is being performed. Whatever the reason, it'd be nice if your deploy process recognized that this server is disabled and that it should not be put back into active use in the load balancer. We'll go over how to do this using the Warthog library below.

First, create the client instance that we'll be using.

```
from warthog.api import WarthogClient, STATUS_DISABLED

client = WarthogClient('https://lb.example.com', 'deploy', 'my password')
```

Next, we'll check the current status of the node when deploying to it.

```
already_disabled = STATUS_DISABLED == client.get_status('app1.example.com')
```

If the server was already disabled when we found it, we don't need to disable it before deploying to it.

```
if not already_disabled:
    client.disable_server('app1.example.com')

# Your deploy process goes here...

if not already_disabled:
    client.enable_server('app1.example.com')
```

You can see above that:

- If the server was *disabled* when we found it, we didn't disable it before deploying and we didn't enable it after deploying.
- If the server was *enabled* when we found it, we disabled it before deploying and enabled it afterwards.

Suppressing SSL Warnings on older Python Versions

Warning: Disabling SSL warnings is done at your own risk.

Warthog depends on the Requests HTTP library. The Requests library in turn uses the Urllib3 library. If you are using Warthog on Python 2.6 or a version of Python 2.7 prior to 2.7.9, Urllib3 will emit warnings due to a missing SSL-related class. While this is a legitimate warning, it may be the case that you can't upgrade the Python version you are using and would rather not deal with the warning output. If that's the case, you can do something like the following.

```python
import warnings
from requests.packages.urllib3.exceptions import InsecurePlatformWarning

warnings.filterwarnings("ignore", category=InsecurePlatformWarning)

from warthog.api import WarthogClient

client = WarthogClient('https://lb.example.com', 'deploy', 'my password')
```

You can find more documentation about the implications of this in the urllib3 docs.

Summary

Hopefully, these use cases and examples will give you a good idea of how to incorporate the Warthog library into your deploy process.

Library

The public API of the Warthog library is maintained in the `warthog.api` module. This is done for the purposes of clearly identifying which parts of the library are public and which parts are internal.

Functionality in the *`warthog.client`*, *`warthog.config`*, *`warthog.transport`*, and *`warthog.exceptions`* modules is included in this module under a single, flat namespace. This allows a simple and consistent way to interact with the library.

Note: When using the library, always make sure to access classes and functions through the `warthog.api` module, not each individual module.

warthog.client

Simple interface for a load balancer with retry logic and intelligent draining of nodes.

class `warthog.client.`**`WarthogClient`**(*scheme_host*, *username*, *password*, *verify=None*, *ssl_version=None*, *network_retries=None*, *commands=None*)

Client for interacting with an A10 load balancer to get the status of nodes managed by it, enable them, and disable them.

This class is thread safe.

Changed in version 0.8.0: Removed .disabled_context() method.

`__init__`(*scheme_host*, *username*, *password*, *verify=None*, *ssl_version=None*, *network_retries=None*, *commands=None*)

Set the load balancer scheme/host/port combination, username and password to use for connecting and authenticating with the load balancer.

Whether or not to verify certificates when using HTTPS may be toggled via the `verify` parameter. This can enable you to use a self signed certificate for the load balancer while still using HTTPS.

The version of SSL or TLS to use may be specified as a `ssl` module protocol constant via the `ssl_version` parameter.

The maximum number of times to retry network operations on transient errors can be specified via the `network_retries` parameter.

If the command factory is not supplied, a default instance will be used. The command factory is responsible for creating new `requests.Session` instances to be used by each command. It is typically only necessary to override this for unit testing purposes.

Changed in version 0.9.0: Added the optional `verify` parameter to make use of self-signed certs easier.

Changed in version 0.10.0: Added the optional `ssl_version` parameter to make use of alternate SSL or TLS versions easier.

Changed in version 2.0.0: Added the optional `network_retries` parameter to make use of retry logic on transient network errors. A non-zero number of retries is used by default if this is not specified. Previously, no retries were attempted on transient network errors.

Changed in version 2.0.0: Removed the optional `wait_interval` parameter. This is now passed directly as an argument to `enable_node()` or `disable_node()` methods.

Parameters

- **`scheme_host`** (*`basestring`*) – Scheme, host, and port combination of the load balancer.
- **`username`** (*`basestring`*) – Name of the user to authenticate with.
- **`password`** (*`basestring`*) – Password for the user to authenticate with.
- **`verify`** (*`bool | None`*) – `True` to verify certificates when using HTTPS, `False` to skip verification, `None` to use the library default. The default is to verify certificates.
- **`ssl_version`** (*`int | None`*) – `ssl` module constant for specifying which version of SSL or TLS to use when connecting to the load balancer over HTTPS, `None` to use the library default. The default is to use TLSv1.2.
- **`network_retries`** (*`int | None`*) – Maximum number of times to retry network operations on transient network errors. Default is to retry network operations a non-zero number of times.
- **`commands`** (`CommandFactory`) – Factory instance for creating new commands for starting and ending sessions with the load balancer.

disable_server(*server*, *max_retries=5*, *wait_interval=2.0*)

Disable a server at the node level, optionally retrying when there are transient errors and waiting for the number of active connections to the server to reach zero.

If `max_retries` is zero, no attempt will be made to wait until there are no active connections to the server, the method will try a single time to disable the server and then return immediately.

Changed in version 2.0.0: Added the optional `wait_interval` parameter.

Parameters

- **server** (*basestring*) – Hostname of the server to disable
- **max_retries** (*int*) – Max number of times to sleep and retry while waiting for the number of active connections to a server to reach zero.
- **wait_interval** (*float*) – How long (in seconds) to wait between each check to see if the number of active connections to a server has reached zero.

Returns True if the server was disabled, false otherwise.

Return type bool

Raises

- ***warthog.exceptions.WarthogAuthFailureError*** – If authentication with the load balancer failed when trying to establish a new session for this operation.
- ***warthog.exceptions.WarthogNoSuchNodeError*** – If the load balancer does not recognize the given hostname.
- ***warthog.exceptions.WarthogApiError*** – If there are any other problems disabling the given server.

enable_server(*server*, *max_retries=5*, *wait_interval=2.0*)

Enable a server at the node level, optionally retrying when there are transient errors and waiting for the server to enter the expected, enabled state.

If `max_retries` is zero, no attempt will be made to wait until the server enters the expected, enabled state, the method will try a single time to enable the server then return immediately.

Changed in version 2.0.0: Added the optional `wait_interval` parameter.

Parameters

- **server** (*basestring*) – Hostname of the server to enable
- **max_retries** (*int*) – Max number of times to sleep and retry while waiting for the server to enter the “enabled” state.
- **wait_interval** (*float*) – How long (in seconds) to wait between each check to see if the server has entered the “enabled” state.

Returns True if the server was enabled, false otherwise

Return type bool

Raises

- ***warthog.exceptions.WarthogAuthFailureError*** – If authentication with the load balancer failed when trying to establish a new session for this operation.
- ***warthog.exceptions.WarthogNoSuchNodeError*** – If the load balancer does not recognize the given hostname.

- ***warthog.exceptions.WarthogApiError*** – If there are any other problems enabling the given server.

get_connections(*server*)
Get the current number of active connections to a server, at the node level.

The number of connections will be 0 or a positive integer.

Parameters **server** (*basestring*) – Hostname of the server to get the number of active connections for.

Returns The number of active connections total for the node, across all groups the server is in.

Return type int

Raises

- ***warthog.exceptions.WarthogAuthFailureError*** – If authentication with the load balancer failed when trying to establish a new session for this operation.
- ***warthog.exceptions.WarthogNoSuchNodeError*** – If the load balancer does not recognize the given hostname.
- ***warthog.exceptions.WarthogApiError*** – If there are any other problems getting the active connections for the given server.

New in version 0.4.0.

get_status(*server*)
Get the current status of the given server, at the node level.

The status will be one of the constants `warthog.core.STATUS_ENABLED` `warthog.core.STATUS_DISABLED`, or `warthog.core.STATUS_DOWN`.

Parameters **server** (*basestring*) – Hostname of the server to get the status of.

Returns The current status of the server, enabled, disabled, or down.

Return type basestring

Raises

- ***warthog.exceptions.WarthogAuthFailureError*** – If authentication with the load balancer failed when trying to establish a new session for this operation.
- ***warthog.exceptions.WarthogNoSuchNodeError*** – If the load balancer does not recognize the given hostname.
- ***warthog.exceptions.WarthogApiError*** – If there are any other problems getting the status of the given server.

class `warthog.client.`**CommandFactory**(*transport_factory*)
Factory for getting new `warthog.core` command instances that each perform some type of request against the load balancer API.

It is typically not required for user code to instantiate this class directly unless you have special requirements and need to inject a custom `transport_factory` method.

This class is thread safe.

__init__(*transport_factory*)
Set the a factory that will create new HTTP Sessions instances to be used for executing commands.

Parameters **transport_factory** (*callable*) – Callable for creating new Session instances for executing commands.

`get_active_connections`(*scheme_host*, *session_id*, *server*)
Get a new command to get the number of active connections to a server.

Parameters

- **`scheme_host`** (*`basestring`*) – Scheme, host, and port combination of the load balancer.
- **`session_id`** (*`basestring`*) – Previously authenticated session ID.
- **`server`** (*`basestring`*) – Host name of the server to get the number of active connections to.

Returns A new command to get active connections to a server.

Return type warthog.core.NodeActiveConnectionsCommand

`get_disable_server`(*scheme_host*, *session_id*, *server*)
Get a new command to disable a server at the node level.

Parameters

- **`scheme_host`** (*`basestring`*) – Scheme, host, and port combination of the load balancer.
- **`session_id`** (*`basestring`*) – Previously authenticated session ID.
- **`server`** (*`basestring`*) – Host name of the server to disable.

Returns A new command to disable a server.

Return type warthog.core.NodeDisableCommand

`get_enable_server`(*scheme_host*, *session_id*, *server*)
Get a new command to enable a server at the node level.

Parameters

- **`scheme_host`** (*`basestring`*) – Scheme, host, and port combination of the load balancer.
- **`session_id`** (*`basestring`*) – Previously authenticated session ID.
- **`server`** (*`basestring`*) – Host name of the server to enable.

Returns A new command to enable a server.

Return type warthog.core.NodeEnableCommand

`get_server_status`(*scheme_host*, *session_id*, *server*)
Get a new command to get the status (enabled / disabled) of a server.

Parameters

- **`scheme_host`** (*`basestring`*) – Scheme, host, and port combination of the load balancer.
- **`session_id`** (*`basestring`*) – Previously authenticated session ID.
- **`server`** (*`basestring`*) – Host name of the server to get the status of.

Returns A new command to get the status of a server.

Return type warthog.core.NodeStatusCommand

`get_session_end`(*scheme_host*, *session_id*)
Get a new command instance to close an existing session.

Parameters

- **scheme_host** (*basestring*) – Scheme, host, and port combination of the load balancer.
- **session_id** (*basestring*) – Previously authenticated session ID.

Returns A new command to close a session.

Return type warthog.core.SessionEndCommand

get_session_start(*scheme_host*, *username*, *password*)

Get a new command instance to start a session.

Parameters

- **scheme_host** (*basestring*) – Scheme, host, and port combination of the load balancer.
- **username** (*basestring*) – Name of the user to authenticate with.
- **password** (*basestring*) – Password for the user to authenticate with.

Returns A new command to start a session.

Return type warthog.core.SessionStartCommand

warthog.config

Load and parse configuration for a client from an INI-style file.

class `warthog.config.`**WarthogConfigLoader**(*config_file=None*, *encoding=None*, *path_resolver=None*, *config_parser=None*)

Load and parse configuration from an INI-style WarthogClient configuration file.

If a specific configuration file is given during construction, this file will be used instead of checking the multiple possible default locations for configuration files. The default locations to be checked are an ordered list of paths contained in `DEFAULT_CONFIG_LOCATIONS`.

Note: When checking for configuration files in default locations, each file will only be checked to see if it exists. It will not be checked to see if the file is readable or correctly formatted.

This class is thread safe.

New in version 0.4.0.

Changed in version 0.6.0: Loading, parsing, and access of configuration settings is now thread safe.

Changed in version 0.6.0: The .parse_configuration() method has been removed and the functionality has been split into the .initialize() and .get_settings() methods.

Changed in version 0.10.0: The `WarthogConfigLoader.__init__()` method no longer directly takes a standard library INI parser as an option parameter, instead it now takes a WarthogConfigParser instance as an optional parameter.

Changed in version 0.10.0: See *Changelog* or *CLI Tool* for details about the changes to configuration file format.

__init__(*config_file=None*, *encoding=None*, *path_resolver=None*, *config_parser=None*)

Optionally, set a specific configuration file, the encoding of the file, resolver to determine the configuration file to use, and custom configuration parser implementation.

By default, multiple locations will be checked for a configuration file, the file is assumed to use UTF-8 encoding, and an

Parameters

- **`config_file`** (*`str|unicode`*) – Optional explicit path to a configuration file to use.
- **`encoding`** (*`str|unicode`*) – Encoding to use for reading the configuration file. Default is UTF-8
- **`path_resolver`** (*`callable`*) – Callable that accepts a single argument (the explicit configuration file path to use) and determines what configuration file to use. It is typically only necessary to set this parameter for unit testing purposes.
- **`config_parser`** (*`WarthogConfigParser`*) – Optional configuration parser to use for reading and parsing the expected INI format for a Warthog configuration file. It is typically only necessary to set this parameter for unit testing purposes.

`get_settings``()`

Get previously loaded and parsed configuration settings, raise an exception if the settings have not already been loaded and parsed.

New in version 0.6.0.

Returns Struct of configuration settings for the Warthog client

Return type *WarthogConfigSettings*

Raises **`RuntimeError`** – If a configuration file has not already been loaded and parsed.

`initialize``()`

Load and parse a configuration an INI-style configuration file.

The values parsed will be stored as a *`WarthogConfigSettings`* instance that may be accessed with the *`get_settings()`* method.

New in version 0.6.0.

Changed in version 0.8.0: Errors locating or parsing configuration files now result in Warthog-specific exceptions (*`warthog.exceptions.WarthogConfigError`*) instead of *ValueError*, *IOError*, or *RuntimeError*.

Returns Fluent interface

Return type *WarthogConfigLoader*

Raises

- ***`warthog.exceptions.WarthogNoConfigFileError`*** – If no explicit configuration file was given and there were no configuration files in any of the default locations checked or if the configuration file could not be opened or read for some reason.
- ***`warthog.exceptions.WarthogMalformedConfigFileError`*** – If the configuration file was malformed such has missing the required ‘warthog’ section or any of the expected values. See the *CLI Tool* section for more information about the expected configuration settings.

class `warthog.config.`**`WarthogConfigSettings`**`(`*scheme_host*, *username*, *password*, *verify*, *ssl_version*`)`

`password`

Alias for field number 2

`scheme_host`

Alias for field number 0

ssl_version
Alias for field number 4

username
Alias for field number 1

verify
Alias for field number 3

warthog.transport

Methods to configure how to interact with the load balancer API over HTTP or HTTPS.

`warthog.transport.`**`get_transport_factory`**(*verify=None*, *ssl_version=None*, *retries=None*)
Get a new callable that returns `requests.Session` instances that have been configured according to the given parameters.

`requests.Session` instances are then used for interacting with the API of the load balancer over HTTP or HTTPS.

Changed in version 0.10.0: Using the requests/urllib3 default is no longer an option. Passing a `None` value for `ssl_version` will result in using the Warthog default (TLS v1).

Changed in version 2.0.0: Added the `retries` parameter and default it to a number greater than zero.

Parameters

- **verify** (*bool | None*) – Should SSL certificates by verified when connecting over HTTPS? Default is `True`. If you have chosen not to verify certificates warnings about this emitted by the requests library will be suppressed.
- **ssl_version** (*int | None*) – Explicit version of SSL to use for HTTPS connections to an A10 load balancer. The version is a constant as specified by the `ssl` module. The default is TLSv1.
- **retries** (*int | None*) – The maximum number of times to retry operations on transient network errors. Note this only applies to cases where we haven't yet sent any data to the server (e.g. connection errors, DNS errors, etc.)

Returns A callable to return new configured session instances for making HTTP(S) requests

Return type callable

warthog.exceptions

Exceptions raised by the Warthog client or library.

exception `warthog.exceptions.`**`WarthogApiError`**(*msg*, *api_msg=None*, *api_code=None*)
Bases: *`warthog.exceptions.WarthogError`*

Base for errors raised in the course of interacting with the load balancer.

`__init__`(*msg*, *api_msg=None*, *api_code=None*)

exception `warthog.exceptions.`**`WarthogAuthFailureError`**(*msg*, *api_msg=None*, *api_code=None*)
Bases: *`warthog.exceptions.WarthogApiError`*

The credentials for authentication are invalid.

exception `warthog.exceptions.`**`WarthogConfigError`**(*msg*)

Bases: *`warthog.exceptions.WarthogError`*

Base for errors raised while parsing or loading configuration.

exception `warthog.exceptions.`**`WarthogError`**(*msg*)

Bases: `exceptions.Exception`

Base for all errors raised by the Warthog library.

`__init__`(*msg*)

exception `warthog.exceptions.`**`WarthogInvalidSessionError`**(*msg*, *api_msg=None*, *api_code=None*)

Bases: *`warthog.exceptions.WarthogApiError`*

The session ID or auth token used while performing some action is unrecognized.

exception `warthog.exceptions.`**`WarthogMalformedConfigFileError`**(*msg*, *missing_section=None*, *missing_option=None*)

Bases: *`warthog.exceptions.WarthogConfigError`*

The configuration file is missing required sections or fields.

`__init__`(*msg*, *missing_section=None*, *missing_option=None*)

exception `warthog.exceptions.`**`WarthogNoConfigFileError`**(*msg*, *locations_checked=None*)

Bases: *`warthog.exceptions.WarthogConfigError`*

No configuration file could be found.

`__init__`(*msg*, *locations_checked=None*)

exception `warthog.exceptions.`**`WarthogNoSuchNodeError`**(*msg*, *api_msg=None*, *api_code=None*, *server=None*)

Bases: *`warthog.exceptions.WarthogNodeError`*

The host being operated on is unrecognized.

exception `warthog.exceptions.`**`WarthogNodeError`**(*msg*, *api_msg=None*, *api_code=None*, *server=None*)

Bases: *`warthog.exceptions.WarthogApiError`*

Base for errors specific to operating on some individual node.

`__init__`(*msg*, *api_msg=None*, *api_code=None*, *server=None*)

exception `warthog.exceptions.`**`WarthogNodeStatusError`**(*msg*, *api_msg=None*, *api_code=None*, *server=None*)

Bases: *`warthog.exceptions.WarthogNodeError`*

There was some error while getting the status of a node.

exception `warthog.exceptions.`**`WarthogPermissionError`**(*msg*, *api_msg=None*, *api_code=None*, *server=None*)

Bases: *`warthog.exceptions.WarthogNodeError`*

The credentials lack required permissions to perform an operation.

New in version 1.999.0.

CLI Tool

Since version *0.4.0* Warthog includes a CLI client for using to interact with a load balancer in addition to the library client. The CLI client supports most of the same operations as the *`warthog.client.WarthogClient`* class. The CLI client has the advantage of working from any type of deploy process, not just Python ones.

Configuration and usage of the CLI client is described below.

Usage

Usage of the Warthog CLI client is based on running various commands. Commands are specified as arguments to the `warthog` program. Some of the commands take additional arguments or options. The most basic usage looks like the following.

```
$ warthog default-config
```

In the example above, we run the `default-config` command with no options or arguments. The `default-config` simply prints an example configuration file for the client.

A slightly more involved example might involve passing an argument or option to the command. The example below explores this.

```
$ warthog status --help
```

The this example passes the `--help` option to the `status` command. This causes the Warthog CLI client to print out the details on using the `status` command.

```
$ warthog status app1.example.com
```

This example passes the argument `app1.example.com` to the `status` command. In this case, the CLI client prints the status of `app1.example.com` (something like `enabled` or `disabled`).

Command-line Options

The following options are supported by the root `warthog` script. For more information about a specific command, use the `--help` option with that command.

`--help`
Display basic usage information about the Warthog CLI client and exit.

`--version`
Display the version of Warthog and exit.

`--config` `<file>`
Set an explicit path to the configuration file for the Warthog CLI client instead of relying on the multiple default locations that are searched. If this option is specified all other potential locations for files are ignored.

`--enable-platform-warning`
Enable warnings from underlying libraries when running on older Python versions (2.6 or 2.7 < 2.7.9) known to cause intermittent failures of SSL/TLS connections. The default is to suppress these warnings.

Commands

`status` `<server>`
Get the status of the given server (by host name). The status will be one of `enabled`, `disabled`, or `down`. If

the server is not in any load balancer pools an error message will be displayed instead and the exit code will be non-zero.

Example:

```
$ warthog status app1.example.com
enabled
```

connections `<server>`

Get the number of active connections to the given server (by host name). The number of active connections will be an integer greater than or equal to zero. If the server is not in any load balancer pools an error message will be displayed instead and the exit code will be non-zero.

Example:

```
$ warthog connections app1.example.com
42
```

disable `<server>`

Disable the given server (by host name). The CLI client will wait until the number of active connections to the server reaches zero before returning. If the server is not in any load balancer pools or was not able to be disabled before the CLI client gave up waiting an error message will be displayed and the exit code will be non-zero. The number of retries attempted is governed by the default value in *`warthog.client.WarthogClient.disable_server()`*.

Example:

```
$ warthog disable app1.example.com
```

enable `<server>`

Enable the given server (by host name). The CLI client will wait until the the server enters the `enabled` state. If the server is not in any load balancer pools or did not enter the `enabled` state before the CLI client gave up waiting an error message will be displayed and the exit code will be non-zero. The number of retires attempted is governed by the default value in *`warthog.client.WarthogClient.enable_server()`*.

Example:

```
$ warthog enable app1.example.com
```

default-config

Print the contents of an example INI-style configuration file for the Warthog CLI client. The output from this command can be piped into a file and then edited for your particular load balancer host and credentials.

Example:

```
$ warthog default-config
[warthog]
scheme_host = https://lb.example.com
username = username
password = password
verify = yes
ssl_version = TLSv1
```

config-path

Print (one path per line) each of the various locations that a configuration file will be searched for if not specified with the `--config` option.

Example:

```
$ warthog config-path
/etc/warthog/warthog.ini
/etc/warthog.ini
/usr/local/etc/warthog/warthog.ini
/usr/local/etc/warthog.ini
/home/user/.warthog.ini
/home/user/something/warthog.ini
```

Configuration

Up till now we've mentioned that the Warthog CLI client uses a configuration file but we haven't really gotten into what exactly that configuration file is or what it looks like. Let's go over that now.

In order to interact with your load balancer over the HTTP or HTTPS API, the Warthog client needs a few pieces of information.

- The scheme, host (or IP), and port that it should use for talking to the load balancer.
- The username it should use for authentication with the load balancer.
- The password associated with the username it should use.
- Whether or not SSL certificates should be validated (similar to how your browser validates them) if using HTTPS.
- The version of SSL / TLS to use if using HTTPS.

Syntax

The Warthog CLI client uses an INI-style configuration file. The format is shown below.

```
[warthog]
scheme_host = https://lb.example.com
username = username
password = password
verify = yes
ssl_version = TLSv1
```

`scheme_host`	Combination of scheme (either 'http' or 'https'), host name (or IP), and port number. This is used to connect to the
`username`	The username to use for authentication with the load balancer. Some examples of valid values: `admin`, `deploy`,
`password`	Password to use along with the username for authentication with the load balancer. This setting is required.
`verify`	If connecting to the load balancer over HTTPS, boolean to indicate if the SSL certificate should be validated. This
`ssl_version`	SSL / TLS version to use when connecting to the load balancer over HTTPS. This version must be supported by yo

Changed in version 0.10.0: The `verify` parameter is now optional. If not specified the Warthog library default will be used (`True` to verify certificates).

Changed in version 0.10.0: The `ssl_version` parameter is now supported and optional. If not specified the Warthog library default will be used (TLSv1).

Location

If the `--config` option is not given to the Warthog CLI client, several locations will be checked for a configuration file to use. The logic for deciding which locations to check is described below. The locations will be checked in order until one that exists is found.

Note: Searching for a configuration file will stop after the first one that exists, NOT the first one that can be read and contains valid values.

1. `/etc/warthog/warthog.ini`
2. `/etc/warthog.ini`
3. `$PREFIX/etc/warthog/warthog.ini` where $PREFIX is the value of `sys.prefix` in Python
4. `$PREFIX/etc/warthog.ini` where $PREFIX is the value of `sys.prefix` in Python
5. `$HOME/.warthog.ini` where $HOME is the home directory of the user running the script
6. `$CWD/warthog.ini` where $CWD is the current working directory when the script is run

If none of these paths exist and the `--config` option is not given, the CLI client will abort.

Development

Have a great idea for a new feature? Want to fix a bug? Then, this guide is for you!

Prerequisites

These instructions assume that you have the following things available.

- Python 2.6 - 2.7 or Python 3.3 - Python 3.6
- The virtualenv tool
- The pip tool
- Git

Environment Setup

First, fork the Warthog Client on GitHub.

Check out your fork of the source code.

```
$ git clone https://github.com/you/warthog.git
```

Create and set up a branch for your awesome new feature or bug fix.

```
$ cd warthog
$ git checkout -b feature-xyz
$ git push origin feature-xyz:feature-xyz
$ git branch -u origin/feature-xyz
```

Set up a virtual environment.

```
$ virtualenv env
```

Enter the virtualenv install required dependencies.

```
$ source env/bin/activate
$ pip install -r requirements.txt
$ pip install -r requirements-dev.txt
```

Install the checkout in "development mode".

```
$ pip install -e .
```

Contributing

Next, code up your feature or bug fix and create a pull request. Some things to keep in mind when submitting a pull request are listed below.

- Your pull request should be a single commit that adds only a single feature or fixes only a single bug. Take a look at this post that describes squashing all your commits into only a single commit, or, the Github documentation on rebasing.
- If you've added a new feature, it should have unit tests. If you've fixed a bug, it should *definitely* have unit tests.
- Your code should follow the Python Style Guide and/or match the surrounding code.

If you're new to Git or GitHub, take a look at the GitHub help site.

Please keep in mind however, that since the Warthog Client has a very specific purpose, pull requests that are determined to be out of scope may not be merged.

Useful Commands

As you do development on the Warthog Client, the following commands might come in handy. Note that all these commands assume you are in the context of the virtualenv you set up earlier.

Build the documentation.

```
$ fab clean docs
```

Run the unit tests for Python 2.7, 3.3, and 3.4.

```
$ tox test
```

Run the unit tests for a specific Python version.

```
$ TOXENV=py27 tox test
```

Run the PyLint tool to find bugs or places where best practices are not being followed.

```
$ fab lint
```

Check how much of the code in the Warthog client is covered by unit tests.

```
$ fab coverage
```

Releasing

Note: The intended audience for this section is the Warthog library maintainers. If you are a user of the Warthog library, you don't need worry about this.

These are the steps for releasing a new version of the Warthog library. The steps assume that all the changes you want to release have already been merged to the `master` branch. The steps further assumed that you've run all the unit tests and done some ad hoc testing of the changes.

Versioning

The canonical version number for the Warthog library is contained in the file `warthog/__init__.py` (relative to the project root). Increment this version number based on the nature of the changes being included in this release.

Do not commit.

Change Log

Update the *change log* to include all relevant changes since the last release. Make sure to note which changes are backwards incompatible.

Update the date of the most recent version to today's date.

Commit the version number and change log updates.

Tagging

Create a new tag based on the new version of the Warthog library.

```
$ git tag 0.5.0
```

Push the committed changes and new tags.

```
$ fab push push_tags
```

Building

Clean the checkout before trying to build.

```
$ fab clean
```

Build source and binary distributions and upload them.

```
$ fab pypi
```

Update PyPI

If the package metadata has changed since the last release, login to PyPI and update the description field or anything else that needs it.

https://pypi.python.org/pypi/warthog

Changelog

2.0.1 - 2017-07-20

- Disable SNI related SSL warnings from urllib3 when running the CLI tool on Python 2.6. This can still be enabled with the `--enable-platform-warnings` switch if desired by the caller.

2.0.0 - 2017-06-29

- **Breaking change** - The Warthog library now only supports v3 of the A10 load balancer REST API. If you need to interact with a load balancer that uses v2 of the A10 REST API you are advised to use Warthog 1.1.0.
- **Breaking change** - Removed the `wait_interval` parameter from *`warthog.client.WarthogClient`*. This setting can now be specified directly for calls to `enable_node` and `disable_node` methods.
- **Breaking change** - Removed `WarthogAuthCloseError`, `WarthogNodeDisableError`, `WarthogNodeEnableError`. Places where these exceptions were previously raised will now raise *`warthog.exceptions.WarthogApiError`*.
- Network operations are now retried on transient errors per #17.

1.999.2 - 2017-06-28

Note: This version is only meant to be used internally at SmarterTravel as a transition step between two load balancers. As such, it is not available on PyPI.

- Rename `warthog.core3` to `warthog.core`
- Remove deprecated exceptions in `warthog.exceptions`: `WarthogAuthCloseError`, `WarthogNodeDisableError`, `WarthogNodeEnableError`.

1.999.1 - 2017-04-06

Note: This version is only meant to be used internally at SmarterTravel as a transition step between two load balancers. As such, it is not available on PyPI.

- Add support for new Python SSL constant `PROTOCOL_TLS` to allow negotiation of SSL versions between the client and server.

1.999.0 - 2017-04-05

Note: This version is only meant to be used internally at SmarterTravel as a transition step between two load balancers. As such, it is not available on PyPI.

- Add new `warthog.core3` module for support of A10 V3 APIs in parallel with the existing core module as a transition to the new API per #8.

- Make sure to pass `ssl_version` to client created in CLI tool per #9.
- Default to use of TLS version 1.2 when not otherwise specified.

1.1.0 - 2016-01-21

- Disable `InsecurePlatformWarning` from urllib3 by default for the CLI client since this makes it unusable CentOS / Red Hat 6. This warning can be re-enabled for the CLI client with the `--enable-platform-warning` flag. Note that this does not change the behavior of the Warthog library at all, only the CLI client. Fixes #5.

1.0.0 - 2015-12-21

- This is the first stable release of Warthog. From this point on, all breaking changes will only be made in major version releases. This release is functionally the same as the `0.10.0` release.

0.10.0 - 2015-10-05

- Add option to *`warthog.client.WarthogClient`* to allow alternate SSL or TLS versions to be used easily. Fixes #6.
- **Breaking change** - Passing a `None` value to *`warthog.transport.get_transport_factory()`* for the `ssl_version` parameter will now use the Warthog library default (TLSv1) instead of letting requests/urllib3 pick the default.
- Add support for specifying an optional `ssl_version` parameter in INI configuration files. Fixes #4.
- Change `verify` to be optional in INI configuration files.

0.9.1 - 2015-08-07

- Fix bug where a config file needed to be specified by the CLI client even when not required (such as when displaying help). Fixes #3.

0.9.0 - 2015-06-04

- Replace examples documentation with more in depth usage guide (*Usage*).
- Add documentation for performing a release of the library (*Releasing*).
- Move enabling/disabling certificate verification to the *`warthog.client.WarthogClient`* class so that using self-signed certificates is less of a hassle and requires less code.

0.8.3 - 2015-03-18

- Dependency on requests updated to version 2.6.0.
- Packaging fixes (use `twine` for uploads to PyPI, stop using the setup.py `register` command).
- Minor documentation updates.

0.8.2 - 2015-02-09

- Small documentation fixes.
- Add project logo to documentation.
- Dependency on requests updated to version 2.5.1.

0.8.1 - 2014-12-22

- Fixed small documentation issues and changed change log dates.

0.8.0 - 2014-12-22

- **Breaking change** - Changed errors raised by *`warthog.config.WarthogConfigLoader`* to be subclasses of *`warthog.exceptions.WarthogConfigError`* instead of using errors from the standard library (`ValueError`, `IOError`, `RuntimeError`).
- **Breaking change** - Removed the `warthog.client.WarthogClient.disabled_context` context manager method since the level of abstraction didn't match the rest of the methods in the client.
- **Breaking change** - Removed all command classes in `warthog.core` from the public API (`warthog.api`). Users wishing to use them may do so at their own risk.
- Change all server-specific exceptions to be based on *`warthog.exceptions.WarthogNodeError`*.
- Improve error handling for CLI client when the configuration file contains an invalid load balancer host (or port, etc.).
- Bundled 3rd-party libs moved to the `warthog.packages` package.
- Dependency on requests updated to version 2.5.0.

0.7.0 - 2014-11-24

- **Breaking change** - Changed error hierarchy so that all errors related to interacting with the load balancer now extend from *`warthog.exceptions.WarthogApiError`*. The root error class *`warthog.exceptions.WarthogError`* no longer contains any functionality specific to making API requests to the load balancer.

0.6.0 - 2014-11-20

- **Breaking change** - Removed `warthog.config.WarthogConfigLoader.parse_configuration()` method and split the functionality into two new methods. Additionally, the class is now thread safe.
- Renamed "Usage" documentation section to "Examples".

0.5.0 - 2014-11-03

- **Breaking change** - Changed all command `.send()` methods in `warthog.core` to not take any arguments to given them a consistent interface.
- Examples documentation improvements.

- Various code quality improvements.

0.4.2 - 2014-10-29

- Documentation improvements (*Development*).
- Test coverage improvements in `warthog.cli`.

0.4.1 - 2014-10-23

- Added CLI tool for using the Warthog Client. See *CLI Tool*.
- Added `warthog.client.WarthogClient.get_connections()` method for getting the number of active connections to a server.
- Added Exceptions in `warthog.exceptions` to the public api in `warthog.api`.
- Added config parsing module `warthog.config` and add it to the public api in `warthog.api`.

0.3.1 - 2014-10-17

- Changed `setup.py` script to not require setuptools.

0.3.0 - 2014-10-16

- Added *Install* documentation.
- Changed authentication request (`warthog.core.SessionStartCommand`) to use `POST` instead of `GET` though there doesn't seem to be any actual difference as far as the load balancer API is concerned.

0.2.0 - 2014-10-14

- Added *Design*, Examples, and *Library* documentation.
- Added test to ensure exported API is consistent.

0.1.0 - 2014-10-11

- Initial release

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index