ElasticPyProxy

Release 1.0

May 25, 2020

Contents:

ElasticPyProxy (EP2) is a controller written completely in python for dynamically scaling HAProxy backend servers. Using this controller, it is possible to integrate HAProxy with a server orchestrator which spwans servers dynamically and scales out and in very frequently. As of now it provides support for the following:

- AWS Autoscaling groups
- Consul

however handler for any orchestrator which exposes an API for getting live backends can be added easily.

It is to be noted that **consul is not an orchestrator** but a service discovery tool. It can be used to discovery A given service can be discovered with consul and later the hosts/nodes which are the provider of that service can be discovered by either consul DNS or consul catalog API. If a node providing the given services goes down, the api will remove those nodes from the catalogue and show only the live ones.

In the rest of the documentation we will continue refering to AWS ASG while explaining the differemt features of EP2 but evrything will be applicable to Consul as well.

So, going ahead with aws, it is possible, using EP2 to integrate HAProxy with a AWS Autoscaling Group. Once integrated, the HAProxy backend servers will scale out and in with the ASG of interest. Thus, whenever the ASG spawns a new instance, that instance will get added to haproxy's concerned backend/listener and when the ASG removes a backend, that particular server will also be removed from HAProxy's concerned backend/listener.

Know more about Hashicorp Consul here

In the rest of the documentation, for simplicity the term **orchestrator** will be used to refer to AWS ASG and consul (although consul is not an orchestrator as already mentioned above but a service discovery mechanism, any orchestrator can be exposed via consul, even AWS ASG) and the backend servers will be refered to as just **backends**

View EP2 on Github

CHAPTER 1

How EP2 works

Simple put EP2, continuously polls the orchestrator and checks what are the available backends and updates haproxy accordingly. However it can be made to do this simple job in more than one way as needed by the user or the host system. Following are the main tasks done by the components present in EP2

1.1 EP2 working

- The system where EP2 runs should have the HAProxy (v1.8 or above) binary, HAproxy UNIX socket exposed and accessible, optionally systemd service file properly configured.
- When EP2 starts, the first thing it does is bootstrap the controller. The bootstrapping includes creating clients for accessing the orchestrator, making the first call to orchestrator API for getting current live backends, updating the haproxy config file using the provided template.
- Once the config file has been updated, the bootstrapper checks if HAProxy is already running. If it is already running, the bootstrapper simply reloads HAProxy so that the new configuration takes affect. If Haproxy is stopped the it starts it.
- Once bootstrap is done, we now have a running haproxy with the current live backends added to it. Post this, EP2 enters its poll-update-repeat loop.
- Once EP2 enters the loop, it primarily does two things. Firstly it polls the orchestrator for the current backend nodes. On getting the list of current live backends it compares it against a locally saved in memory list of live backends. If there is a difference, it updates the local in-memory list and goes on to update HAProxy otherwise it does nothing
- EP2 can update haproxy in two ways. First way is, it simply formats the configured haproxy template file with the live backend servers, updates the HAProxy config file with the contents of the formatted template file and reloads HAProxy.
- Since HAProxy reload (post v1.8) is hitless, reload wont cause any downtime.
- EP2 allows two ways to reload HAProxy, one via systemd service and the other via the HAProxy binary. The respective params must be provided in EP2 config accordingly. More on this below.

- The issue with the above method of updating is, HAProxy has to be reloaded. When the number of reloads is less, it is not a big issue. However if the number of reload is too high, it can cause overhead since reload essentially involves transfer of connections/sockets from old process to the new process.
- The second method of updation is the one in which reload is not required at all. It updates HAProxy in runtime using the UNIX socket file it exposes. This is to some extent complicated than the previous method. Once the new backends are added the config file is also updated so that the runtime configuration and the config file on disk remains consistent, but there is no need to reload HAProxy.
- Once updation is done, it waits for a configured amount of time before polling for backends again and repeating the same processes.

1.2 Major components of EP2

- Backend fetcher : The backend fetcher fetches the live backends from the configured orchestrator. As mentioned earlier for now this is AWS ASG and Consul.
- HaproxyUpdater : This updates the HAProxy, either by updating config or via socket at runtime.
- ConfigHandler : This is used by HaproxyUpdater to handle the HAProxy config updation
- RuntimeUpdater : This is used by HaproxyUpdater to update HAProxy at runtime via socket.
- HaproxyReloader : This is used to reload HAProxy wither via systemd or via binary.

CHAPTER 2

Backend fetcher

The awsfetcher and the consulfetcher, fetches the available servers in the concerned asg or service respectively. For AWS ASG, boto3 library is used and for Consul, the Consul catalog API is used.

CHAPTER 3

Updating HAProxy via config

As mentioned above, one of the ways HAProxy can be updated using EP2 is via updating its config direcly. In both the updation methods, EP2 is preconfigured with a template HAProxy config (mentioned below).

Once the current live backend servers are available, EP2 formats the template and populates it with the current live backends. Then it replaces the contents of the actually HAProxy config file with the contents of this formatted template file. After this is done it reloads HAProxy either via **systemd** or via **binary**.

Both the path to Haproxy config file and path to the HAProxy template file should be provided in EP2 config.

CHAPTER 4

Updating HAProxy at Runtime

In this method, HAProxy has to be preconfigured with a number of inactive or disabled backend servers. This is taken care of by the **bootstrapper**. When bootstrap runs, apart from creating the live backend servers it also creates a number of inactive dummy backend servers with dummy address.

The number of dummy backend servers to be created is decided by the config param **node_slots**. If the number of live backend servers fetched from the orchestrator is **x**, then the number of dummy inactive servers created is **node_slots - x**.

Now whenever a scale in activity happens, that is the orchestrator removes some of the live servers, EP2 finds out which servers are out of service. It marks them as inactive and adds them to the inactive pool.

Whenever a new server is spawned up, EP2 picks an inactive server from the pool, changes its address to the address of the newly spawned backend server and marks it as ready. Thus the inactive server now becomes active and it represents the newly spawned backend server.

Once the runtime config of HAProxy has been updated, same configuration is replicated in the config file so that it stays at par with the running config of HAProxy.

It is worth noting that in this procedure, the value of the **node_slots** param should always be greater than the total amount of live servers the orchestrator can contain/spawn at any given time. This should be easily figured out from the **min/max** criteria of the orchestrator in use.

CHAPTER 5

Reloading HAProxy via systemd

When updating HAProxy via config, HAProxy has to be reloaded and one such way to reload HAProxy is via **systemd**. For this there should be a properly configured systemd service file such that systemd reload works properly.

The command used is the usual systemd command:

```
systemctl reoad [haproxy_servicefile_name]
```

The HAProxy systemd service file name should be provided as a EP2 config param.

CHAPTER 6

Reloading HAProxy via binary

The other way to reload haproxy is by executing the binary. For this is to work the following things must be provided in EP2 config :

- haproxy_config_file : The haproxy config file
- haproxy_binary : The location of the HAProxy binary which is usually `/usr/sbin/haproxy`
- haproxy_socket_file : The location of the HAProxy unix socket file.
- pid_file : The location of the HAProxy PID file which is usually `/run/haproxy.pid`

The command fired is the usual one:

```
[haproxy_binary] -W -q -D -f [haproxy_config_file] -p [pid_file] -x [socket_
↪file] -sf $(cat [pid_file])
```

The above causes hitless reload of HAProxy.

CHAPTER 7

Bootstrapping EP2

At the very beginning when EP2 is started, bootstrapping takes place. The following essentially happens in the bootstrap process :

- The desired nodefetcher is initialised. As of now it is the **awsfetcher**. As a part of the initialisation of the awsfetcher, the asg and ec2 boto3 clients are created using the provided aws credentials.
- The the very first call to get the live backend servers is made.
- Once EP2 has the live backend server addresses, irrespective of whether EP2 is configured to use update via config or update at runtime, EP2 updates the haproxy config with the formatted template file contents. It is during this time EP2 creates the inactive pool if it is configured to use updae by runtime on later runs.
- Once the updation is done, it checks whether HAProxy is running or not. If its not running, the it starts HAProxy. If it was running then it simply reloads it using the configured method.
- Once bootstrap is done, EP2 enters its loop.

CHAPTER 8

Installing EP2

EP2 can be installed either using pip or can be built from source.

Installing via pip

Inorder to install via pip, execute the following:

```
sudo pip3 install git+git://github.com/djmgit/ElasticPyProxy
```

Installing from source

Inorder to install from source perform the following actions:

- Clone this repo and enter into it using `git clone https://github.com/djmgit/ElasticPyProxy.git`
- Run the following command `sudo python3 setup.py install`

Once installation is done, ep2 will be installed at **/usr/bin/ep2**

Also the following files and directories will be created:

- /var/log/ep2
- /etc/ep2
- /etc/ep2/ep2.conf
- /etc/ep2/haproxy.cfg.template

CHAPTER 9

Configuring EP2

A sample EP2 config file is given below:

```
[haproxy]
haproxy_config_file = /etc/haproxy/haproxy.cfg
template_file = /home/deep/elasticpyproxy/etc/haproxy.config.template
backend_port = 6003
haproxy_binary = /usr/sbin/haproxy
start_by = systemd
haproxy_socket_file = /var/run/haproxy/haproxy.sock
pid_file = /run/haproxy.pid
backend_name = haproxynode
update_type = update_by_runtime
node_slots = 5
service_name = haproxy
lock_dir = /home/deep/elasticpyproxy/etc
orchestrator = aws
sleep_before_next_run = 5
log_file = /var/log/ep2/ep2.log

[AWS]
aws_access_key_id =
aws_secret_access_key =
asg_name =
region_name =
```

For Consul, the following block can be used instead of [AWS]:

```
[CONSUL]
service_name =
consul_ip =
consul_port =
only_passing =
tags =
```

Params involved:

- haproxy_config_file : This is the path to the actual haproxy config file. Usually it is /etc/haproxy/haproxy.cfg
- template_file : Path to the template file. This is the file that will be populated and used to update the actual haproxy config file.
- backend_port : The port used by backend servers.
- haproxy_binary : The HAProxy binary file location.
- start_by : How to start/reload HAProxy. Can be **systemd** or **binary**
- haproxy_socket_file : Path to HAProxy socket file. If Haproxy has been configure to spawn multiple process via nbproc, then paths to multiple socket files can be provided here separated by comma
- pid_file : Path to HAProxy pid file
- backend_name : The name of the HAProxy backend/listener name under which the live backend servers fetched from orchestrator will be added.
- backend_maxconn : Max connections for individual backends
- check_interval : Interval for performing health checks for individual backends
- update_type : How to update HAProxy. Either **update_by_config** or **udpate_by_runtime**
- node_slots : Total number of slots for backend servers. As mentioned above, this will be used to calculate inactive servers.
- service_name : Service name for HAProxy systemd service. Required only when using reload by systemd
- lock_dir : Path to directory for storing EP2 lock file.
- orchestrator : The backend orchestrator. As of now it can only be **aws**
- sleep_before_next_run : Amount of time to wait before next poll-update run
- log_file : The file to output logs

[AWS]

- aws_access_key_id : aws creds
- aws_secret_access_key : aws creds
- asg_name : Name of the autoscaling group
- region_name : aws region name where the asg exists

[CONSUL]

- service_name : Name of the service which has already been registered with consul and whose providers we want to discover
- consul_ip : IP adress where consul catalog API is running. Default is 127.0.0.1. If the node where EP2 is running has been added the the consul cluster, then consul api should be accessed via 127.0.0.1 if not changed otherwise.
- consul_port : Port for the Consul catalog API. Default is 8500
- only_passing : can be **True** or **False**. If True, then only those backends will be discovered and added for which the service checks are passing. Please refer Consul doc to learn more about service checks. Default value is True.
- tags : Comma separated values of tags to filter services.

A sample haproxy template file is shown below:

```
global
        log /dev/log  local0
        log /dev/log  local1 notice
        chroot /var/lib/haproxy
        stats socket /var/run/haproxy/haproxy.sock mode 660 level admin expose-fd␣
↪listeners
        stats timeout 30s
        user haproxy
        group haproxy
        daemon

        # Default SSL material locations
        ca-base /etc/ssl/certs
        crt-base /etc/ssl/private

        # Default ciphers to use on SSL-enabled listening sockets.
        # For more information, see ciphers(1SSL). This list is from:
        #  https://hynek.me/articles/hardening-your-web-servers-ssl-ciphers/
        # An alternative list with additional directives can be obtained from
        #  https://mozilla.github.io/server-side-tls/ssl-config-generator/?
↪server=haproxy
        ssl-default-bind-ciphers␣
↪ECDH+AESGCM:DH+AESGCM:ECDH+AES256:DH+AES256:ECDH+AES128:DH+AES:RSA+AESGCM:RSA+AES:!
↪aNULL:!MD5:!DSS
        ssl-default-bind-options no-sslv3
        stats socket ipv4@127.0.0.1:9999 level admin
  stats timeout 2m

defaults
        log   global
        mode  http
        option        httplog
        option        dontlognull
        timeout connect 5000
        timeout client  50000
        timeout server  50000
        errorfile 400 /etc/haproxy/errors/400.http
        errorfile 403 /etc/haproxy/errors/403.http
        errorfile 408 /etc/haproxy/errors/408.http
        errorfile 500 /etc/haproxy/errors/500.http
        errorfile 502 /etc/haproxy/errors/502.http
        errorfile 503 /etc/haproxy/errors/503.http
        errorfile 504 /etc/haproxy/errors/504.http

listen haproxynode
        bind *:7001
        balance roundrobin
        option forwardfor
        http-request set-header X-Forwarded-Port %[dst_port]
        http-request set-header X-CLIENT-IP %[src]
        http-request add-header X-Forwarded-Proto https if {{ ssl_fc }}
        option httpchk HEAD / HTTP/1.1\r\nHost:localhost
        {{nodes}}

listen stats
    bind :32700
    stats enable
```

(continues on next page)

(continued from previous page)

```
    stats uri /stat
    stats hide-version
```

The backend/listener used (`haproxynode`) in this case should be mentioned in **EP2 config**. The backend/listener of interest should have the template varibale `nodes` in jinja templating format. This template varibale will be replaced with the live backend servers in each run.

Once this template is formatted, the actual HAProxy config will be updated with the formatted contents of this template file.

So, whatever changes one usually has to make to HAProxy config, they have to be made here.

CHAPTER 10

Starting EP2

Execute the following for starting EP2:

```
sudo ep2 -f [Path to ep2.conf]
```

Stop EP2 by CTRL+C.

The ideal way to run EP2 would be to use a process manager like **systemd** or **supervisord**.

CHAPTER 11

ElasticPyProxy Code Documentation

11.1 driver (Main entry point for ElasticPyProxy)

`src.driver.driver.drive()`

Method for starting ep2

This is the entry method which starts ep2 controller. It calls bootstrap module for bootstrapping ep2, reads ep2 config, initialises haproxy and starts the **poll-update-repeat loop**

Returns Returns nothing

Return type None

11.2 drivercache : Cache layer for EP2

class `src.driver.drivercache.DriverCache(node_ips)`

Class to provide caching for ep2

The backends fetched in a given run is stored in memory. The backends fetched in next run witll be compared to the ones already held by this class (node_ips). If there is a mismatch, only then update will be done

Parameters **node_ips** (*list*) – list of backend IPs

need_to_update(*node_ips*)

Method to check if haproxy needs to be updated

Parameters **node_ips** (*list*) – list of backend IPs

Returns Whether to update haproxy or not

Return type bool

11.3 bootstrap : Bootstrapper for ep2

`src.driver.bootstrap.bootstrap(**kwargs)`

Method to bootstrap EP2 controller

This method bootstraps EP2 to creates the neccessary objects and returns it to the driver.

Parameters ****kwargs** (*object*) – kwargs must contains config dictionary, logger object.

Returns Whether bootstrap updater was successfull or not src.core.haproxyupdater.haproxyupdate.HaproxyUPdate: Object for updating haproxy config src.core.nodefetchers.basefetcher: Object for fetching backends

Return type bool

11.4 haproxyupdate : Module for updating haproxy

class `src.core.haproxyupdater.haproxyupdate.HaproxyUpdate(**kwargs)`

Class for handling haproxy update and reload

This class contains handlers which controls haproxy uptation and reload. Haproxy can be updated wither by updating its config file followed by a reload via systemd or via binary. The other way to reload haproxy is via the exposed socket. This type of update does not require any reload

For updating via runtime haproxy needs to maintain a pool if inactive backends. When a new live backend comes, we can pull an inactive live backend and make it live changing its ip to that of the live backend

Parameters ****kwargs** (*dictionary*) – params in key/value dict format

logger = None

Valid methods to start haproxy.

Note: init methods is not supported yet.

---

update_haproxy()

Updates haproxy config

This method updates haproxy config with the help of the util methods.

Returns Wether haproxy was updated successfully or not.

Return type bool

update_haproxy_by_config_reload(*update_only=False*)

Method to update haproxy via config reload

This method will update haproxy via updating its config and subsequently reloading it. The actual update will be done by the confighandler module and reload will be done by haproxyreloader. Optinaly is **upate_only** is set to True then only config will be updated and reload will not be done.

Parameters **update_only** (*bool*) – Whether only update is required or both update and reload is required.

Returns Whether successfully updated/reloaded as the case may be

Return type bool

update_node_list (*node_list*)
Method to update active node list

This method will e called to update the list of active backends. Haproxy needs to be updated and optionally reloaded if this list changes

Parameters **node_list** (*list*) – List containing IPs/Hostnames of active backends

valid_start_by = None
Valid methods to update haproxy

11.5 confighandler : Module for updating haproxy

class src.core.haproxyupdater.confighandler.**ConfigHandler**
Class to handler haproxy config file updation

This class contains method for updating the haproxy config file with the provided formatted haproxy config template.

The template is first populated with the fetched backends using jinja templating engine and then the haproxy config file is updated with this formatted template.

Parameters ***kwargs** (*dictionary*) – Dictionary containing params

static read_write_file (***kwargs*)
Method to read and write haproxy config file

Parameters ***kwargs** (*dictionary*) – Dictionary containing params

Returns successfully updated or not str : error string if any

Return type bool

static update_config (***kwargs*)
Method for updating haproxy config

This is the method which actually updates the haproxy config file using the provided template file after properly formatting it

Parameters ***kwargs** (*dictionary*) – Dictionary containing params

Returns Successfully updated or not

Return type bool

11.6 haproxyreloader : Module for reloading haproxy

class src.core.haproxyupdater.haproxyreloader.**HaproxyReloader**
Class for handling haproxy reload

This class provides methods to reload haproxy, either via systemd or via the binary.

In order to reload via bianry, the socket file and the PID file should be present as params along with the binary location.

For systemd, the systemd service name should be provided as param.

Both reload via systemd and reload via binary are done by execting shell commands via subprocess library

static reload_haproxy(***kwargs*)

Method for reloading haproxy

Method for reloading haproxy. This takes the help of util method to reload haproxy either via systemd or binary.

Other classes and methods will call this method for updating haporoxy with the required param.

Parameters ****kwargs** (*dictionary*) – Dictionary conatining params

Returns Successfully reloaded or not

Return type bool

static start_by_systemd(*service_name*, *logger=None*)

Method for starting haproxy via systemd

Starts haproxy via systemd. Executes systemd start as a shell command.

Parameters **logger** (*object*) – logger object for logging

Returns Successfully started or not

Return type bool

11.7 runtimeupdater : Module for updating haproxy at runtime

class src.core.haproxyupdater.runtimeupdater.**RuntimeUpdater**

Class for updating haproxy at runtime

This class conatins methods for updating haproxy backends at runtime without reloading it.

This is done by communicating with haproxy over the unix socket file expsed by it.

Once ep2 gets the ips/hostnames of the live backends, it communicates with haproxy over socket, extracts servers from inactive pool and updating their address with that of the live ones.

static update_haproxy_runtime(***kwargs*)

Method to update haproxy at runtime using the util method present above

Parameters ****kwargs** (*dictionary*) – Dictionary containing params

Returns Successfully updated haproxy or not stats : dictionary containing active nodes and inactive node count

Return type bool

static update_runtime_util(*haproxy_sock*, *node_ips*, *nodes*, *backend_name*, *port*, *logger=None*)

Method for updating haproxy backends using unix socket

This method updates the haproxy backends by sending commands over the exposed unix socket.

Working

- First it iterates over the active backends currently present in haproxy.
- If they are not present in the current fetched list of backends, then we disable those backends and add them to the inactive pool.
- Next we iterate over the list of current live nodes fetched from orchestrator
- if they are already present as live backends in haproxy **even after the above elimination** then we skip.

- If they are not present then we fetch a inactive node from the inactive pool, change its address to that of the live node and enable back that node.

Parameters

- **haproxy_sock** (*str*) – Location of the haproxy unix socket file
- **node_ips** (*list*) – List of current live nodes fetched from orchestrator. (IP or hostname)
- **nodes** (*dictionary*) – Dictionary conatining haproxy active and inactive backends
- **backend_name** (*str*) – Name of the haproxy backend that needs to be updated.
- **port** (*int*) – port for the backend nodes
- **logger** (*object*) – Logger object

Retuns: dict : Dictionary conatining active node_ips and inactive nodes count

11.8 sockethandler : Module for handling socket operations

class src.core.haproxyupdater.sockethandler.**SocketHandler**(***kwargs*)

Class containing methods for handling socket operation

This is a generic class for handling all socket operation. All the commands which are to be sent to haproxy and done via methods in this class.

Parameters ****kwargs** (*dictionary*) – Dictionary containing params

connect_socket(*sock_file*)

Method to connect to haproxy unix socket

This method creates a socket connection to the given haproxy unix socket

Returns Successfully created socket connection or not

Return type bool

send_command(***kwargs*)

Method to send command to haproxy unix socket and get response

It will first create a socket connection to the haproxy socket and then send the given command and get response.

Parameters ****kwargs** (*dictionary*) – Dictionary containing params

Returns Successfully sent command or not str : response sent by the haproxy unix socket

Return type bool

11.9 basefetcher : Provides base class for backend fetchers

class src.core.nodefetchers.basefetcher.**BaseFetcher**(***kwargs*)

Base class for all kinds of backend fetchers

This class is the base/super class for all backend fetchers. Individual backend fetchers have to inherit this class and an optionally override the methods present in this class

Parameters ****kwargs** (*dictionary*) – Dictionary containing params

fetch()
Method for fecthing backend nodes from orchestrator

Make request for nodes. Should be overridden by individual fetchers

11.10 orchestrator : Module for providing appropriate backend fetcher

src.core.nodefetchers.orchestrator.**get_orchestrator_handler**(*config*, *logger=None*)
Method for deciding which fetcher to use

Decide which fetcher to use depending on the orchestrator mentioned in the config.

Parameters

- **config** (*dictionary*) – dictionary holding ep2 config
- **logger** (*object*) – logger object

Returns Backend fetcher

Return type object

src.core.nodefetchers.orchestrator.**prepare_aws_handler**(*config*, *logger*)
Prepares the AWS fetcher

Parameters **config** (*dictionary*) – dictionary containing ep2 config

Returns Aws backend fetecher

Return type srv.nodefetchers.awsfetcher.awsfetcher

src.core.nodefetchers.orchestrator.**prepare_consul_handler**(*config*, *logger*)
Prepares the Consul fetcher

Parameters **config** (*dictionary*) – dictionary containing ep2 config

Returns Consul backend fetcher

Return type srv.nodefetchers.consulfetcher.cosulfetcher

11.11 awsfetcher : Module for fecthing live backends from AWS

class src.core.nodefetchers.awsfetcher.awsfetcher.**AwsFetcher**(***kwargs*)
Class for fetching libe backends from AWS

Contains methods to fetch live backends from AWS using the boto3 library To make this class work properly, ep2 config must have aws section with access_key_id and secret_access_token specified along with aws region and ip_type which is required (public or private)

Parameters ****kwargs** (*dictionary*) – Dictionary containing params

fetch()
Method for fetching backends

This method takes help of BotoHandler for fetching backends from AWS and return them to the caller

Returns List of backends

Return type list

11.12 botohandler : Module for handling AWS operations

class src.core.nodefetchers.awsfetcher.botohandler.**BotoHandler**

Class for handling aws node fetching operations

Methods in this class handle aws operations for retreiving the list of active backends. It first uses boto3 asg client for describing the asg of interest. Once we have the instance ids in that asg, we use ec2 client for describing those instances for getting their public/private ips

static get_auto_scaling_client(***kwargs*)

Method for initialising asg boto client

Parameters ****kwargs** (*object*) – kwargs must contains config dictionary, logger object.

Returns boto3 asg client

Return type boto3.client

static get_ec2_client(***kwargs*)

Method for initialising ec2 boto client

Parameters ****kwargs** (*object*) – kwargs must contains config dictionary, logger object.

Returns boto3 ec2 client

Return type boto3.client

static get_instance_ips_for_asg(***kwargs*)

Method for getting aws live instance IPs

Parameters ****kwargs** (*object*) – kwargs must contains config dictionary, logger object.

Returns List of live backend IPs

Return type list

11.13 Indices and tables

- genindex
- modindex
- search

Index

S

U

V