Dyn Documentation

Release 1.8.0

Author

Jul 31, 2017

Contents

1 Introduction **3**
- 1.1 Dyn API License . . . 3

2 Installation **5**
- 2.1 Distribute & Pip . . . 5
- 2.2 Get the Code . . . 5

3 Quickstart **7**
- 3.1 Authentication . . . 7
- 3.2 Managing Your TM Accounts . . . 8
- 3.3 Using your Zones . . . 8
- 3.4 TM Services . . . 9
- 3.5 TM Errors and Exceptions . . . 9
- 3.6 MM Errors and Exceptions . . . 9

4 dyn.tm (Traffic Management) Module **11**
- 4.1 Authentication . . . 11
- 4.2 Zones . . . 14
- 4.3 Accounts . . . 22
- 4.4 Records . . . 35
- 4.5 Services . . . 54
- 4.6 TM Reports . . . 114
- 4.7 TM Tools . . . 115
- 4.8 TM Errors . . . 117

5 dyn.mm (Message Management) Module **119**
- 5.1 MM Accounts . . . 119
- 5.2 Messages . . . 121
- 5.3 Reports . . . 124
- 5.4 MM Session . . . 124
- 5.5 MM Errors . . . 124

6 Advanced Topics **127**
- 6.1 Sessions . . . 127
- 6.2 Password Encryption . . . 129

7 Core **131**

7.1 Singleton 131
7.2 SessionEngine 131

8 Indices and tables **135**

Python Module Index **137**

Release v1.8.0. (*Installation*)

With the latest release of this sdk, it's now even easier to manage both your Dyn Traffic Management and Message Management services.

```
>>> from dyn.tm.session import DynectSession
>>> from dyn.tm.zones import Zone
>>> dynect_session = DynectSession(customer, username, password)
>>> my_zone = Zone('mysite.com')
>>> my_zone.status
'active'
>>> new_rec = my_zone.add_record('maps', 'A', '127.0.0.1')
>>> new_rec.fqdn
'maps.mysite.com.'
>>> my_zone.get_all_records()
{u'a_records': [<ARecord>: 127.0.0.1, <ARecord>: 127.0.1.1]}
```

Contents:

CHAPTER 1

Introduction

The Dyn Python SDK is an Object Oriented API wrapper. This library encompasses all of the functionality provided by both the Dyn Traffic Management and Message Management APIs. For more additional documentation on the Dyn APIs please see the Dyn Developer Resources page.

Dyn API License

CHAPTER 2

Installation

Follow the instructions below to install the Dyn module.

Distribute & Pip

The easiest way to install the Dyn module is via 'pip'.

1. Go to <http://www.pip-installer.org/>', download and install 'pip'.
2. In Terminal, run:

    $ pip install dyn

Get the Code

Dyn is actively developed on GitHub. The code is always available ''<https://github.com/dyninc/dyn-python>' and there are several options available to obtain the code.

Clone the public repository:

```
git clone https://github.com/dyninc/dyn-python.git
```

Download the tarball:

```
$ curl -OL https://github.com/dyninc/dyn-python/tarball/master
```

Download the zipball:

```
$ curl -OL https://github.com/dyninc/dyn-python/zipball/master
```

Once you have a copy of the source, you can embed it in your Python package, or install it into your site-packages by running:

```
$ python setup.py install
```

CHAPTER 3

Quickstart

Eager to get started? This guide will help you get started managing your Dyn services using this module.

If you have not already, *Install* the Dyn module before proceeding further.

It is also important to understand that this library handles interacting with both Traffic Management (TM) and Message Management (MM) services. For both TM and MM, you will need to create Session objects to handle API interactions, processing API responses, and creating the various objects described in the *TM* and *MM* API documentation sections.

Here are some simple examples to get you started.

Authentication

API sessions will need to be created each time you use either of these libraries. These session objects internally manage interaction with the API.

To create a TM DynectSession, begin by importing the tm.session module:

```
>>> from dyn.tm.session import DynectSession
```

Now create an instance of a DynectSession by using our Dyn login credentials:

```
>>> my_session = DynectSession(customer, username, password)
```

Now you have a `DynectSession` object called `my_session`. You will be able to use this to access your available resources.

For MM, you can import and create an `MMSession` from the mm.session module:

```
>>> from dyn.mm.session import MMSession
```

Now create an instance of this session by providing it an API Key:

```
>>> mm_session = MMSession(my_api_key)
```

This object will now grant you access to the features provided by the Email API.

Managing Your TM Accounts

The new wrapper allows you easy access to managing all of the elements within your account, such as new `Users` objects:

```
>>> from dyn.tm.accounts import User
>>> jsmith = User('jsmith')
>>> jsmith.status
u'blocked'
>>> jsmith.unblock()
>>> jsmith.status
u'active'
>>> jsmith.get_permissions_report()
['ZoneAdd', 'ZoneDelete', 'Login']
>>> jsmith.add_permission('ZoneGet')
>>> jsmith.get_permissions_report()
['ZoneAdd', 'ZoneDelete', 'Login', 'ZoneGet']
```

You can also create new `PermissionGroups` that can later be applied to `User` objects

```
>>> from dyn.tm.accounts import PermissionsGroup
>>> sample = PermissionsGroup('Sample', 'Sample permission Group')
>>> sample.add_permissions('DSFAdd')
>>> sample.add_permissions('DSFGet')
>>> sample.add_permissions('DSFDelete')
>>> sample.add_zone('mysite.com')
```

Using your Zones

Using your current session you can create a new zone:

```
>>> from dyn.tm.zones import Zone
>>> my_zone = Zone('mysite.com', 'myemail@email.com')
```

You can also access your previously created zones:

```
>>> my_old_zone = Zone('example.com')
```

Using these `Zone` objects you can begin to manipulate your zones, such as, adding a record:

```
>>> a_rec = my_zone.add_record('node', 'A', '127.0.0.1')
>>> a_rec.ip
u'127.0.0.1'
>>> a_rec.fqdn
u'node.mysite.com.'
>>> a_rec.get_all_records()
{'a_records': [127.0.0.1], 'aaaa_records': [], ...}
```

TM Services

Try adding a `DynamicDNS` service to your zone:

```
>>> ddns = my_zone.add_service(service_type='DDNS', record_type='A',
...                            address='127.0.0.1')
>>> ddns.zone
u'mysite.com'
>>> ddns.active
u'Y'
```

TM Errors and Exceptions

In the event of an authentication problem, dyn.tm will raise a *DynectAuthError* exception.

In the event an error in an API Creation is encountered, dyn.tm will raise a *DynectCreateError* exception with additional information about why the POST failed.

In the event an error in an API Update is encountered, dyn.tm will raise a *DynectUpdateError* exception with additional information about why the PUT failed.

In the event an error in an API Get is encountered, dyn.tm will raise a *DynectGetError* exception with additional information about why the GET failed.

In the event an error in an API Deletion is encountered, dyn.tm will raise a *DynectDeleteError* exception with additional information about why the DELETE failed.

In the event an error in an API request returns with an incomplete status (i.e. the requested job has not yet completed) the wrapper will poll until either the job has completed or the polling times out. In such an event, dyn.tm will raise a *DynectQueryTimeout* exception

All exceptions that dyn.tm explicitly raises inherit from `dyn.tm.errors.DynectError`.

MM Errors and Exceptions

In the event that an invalid API Key is provided to your `MMSession` an *EmailKeyError* exception will be raised.

If you passed an invalid argument to one of the provided MM objects, a *DynInvalidArgumentError* exception is raised.

The *DynInvalidArgumentError* should not be confused with the *EmailInvalidArgumentError*. The latter is raised if a required field is not provided. This is an unlikely exception to be raised as the error would likely be raised as *DynInvalidArgumentError*. However, it is still a possible scenario.

The *EmailObjectError* will be raised if you attempt to create an object that already exists on the Dyn MM system.

All MM exceptions inherit from `EmailError`

Ready for more? Check out the *TM* and *MM* module documentation sections, the full TM API Documentation or the MM API Documentation.

CHAPTER 4

dyn.tm (Traffic Management) Module

The dyn.tm (TM) module provides access to all of the Traffic Management resources provided by Dyn's Traffic Management REST API. It's important to note that all code examples assume the existence of a `DynectSession` instance. This object is used by the modules described below to access the API and make their associated calls. If you are looking for information on a specific function, class or method, this part of the documentation is for you.

Authentication

The `session` module is an interface to authentication via the dyn.tm REST API. As noted in the advanced section, `DynectSession`'s are implemented as Singleton types, which means that, in most cases, you don't need to keep track of your `DynectSession` instances after you create them. However, there are several examples of ways in which you can use these session instances which will be outlined below.

class dyn.tm.session.**DynectSession**(*customer*, *username*, *password*, *host='api.dynect.net'*, *port=443*, *ssl=True*, *api_version='current'*, *auto_auth=True*, *key=None*, *history=False*, *proxy_host=None*, *proxy_port=None*, *proxy_user=None*, *proxy_pass=None*)

Base object representing a DynectSession Session

__init__(*customer*, *username*, *password*, *host='api.dynect.net'*, *port=443*, *ssl=True*, *api_version='current'*, *auto_auth=True*, *key=None*, *history=False*, *proxy_host=None*, *proxy_port=None*, *proxy_user=None*, *proxy_pass=None*)

Initialize a Dynect Rest Session object and store the provided credentials

Parameters

- **host** – DynECT API server address
- **port** – Port to connect to DynECT API server
- **ssl** – Enable SSL
- **api_version** – version of the api to use
- **customer** – DynECT customer name

- **username** – DynECT Customer's username
- **password** – User's password
- **auto_auth** – declare whether or not to automatically log in
- **key** – A valid AES-256 password encryption key to be used when encrypting your password
- **history** – A boolean flag determining whether or not you would like to store a record of all API calls made to review later
- **proxy_host** – A proxy host to utilize
- **proxy_port** – The port that the proxy is served on
- **proxy_user** – A username to connect to the proxy with if required
- **proxy_pass** – A password to connect to the proxy with if required

authenticate()
Authenticate to the DynectSession service with the provided credentials

log_out()
Log the current session out from the DynECT API system

permissions
Permissions of the currently logged in user

update_password(*new_password*)
Update the current users password

Parameters **new_password** – The new password to use

uri_root = '/REST'

user_permissions_report(*user_name=None*)
Returns information regarding the requested user's permission access

Parameters **user_name** – The user whose permissions will be returned. Defaults to the current user

The Basics

For basic usage, you need not do anything more than simply

```
>>> from dyn.tm.session import DynectSession
>>> DynectSession('customer', 'user', 'password')
```

Permissions

Using a *DynectSession* instance, you can also verify the current permissions associated with your session by simply checking the permissions property of your *DynectSession* instance.

```
>>> from dyn.tm.session import DynectSession
>>> s = DynectSession('customer', 'user', 'password')
>>> s.permissions
[u'ZoneGet', u'ZoneUpdate', u'ZoneCreate', u'ZoneDelete', ...
```

Additional Features

The majority of these features exist mainly to provide a cleaner interface to working with sessions as Singleton types.

Multiple Sessions

To manage multiple user accounts, use a `DynectMultiSession` instance and call the *new_user_session* method

```
>>> from dyn.tm.session import DynectMultiSession
>>> s = DynectMultiSession('customer', 'user', 'password')
>>> s.new_user_session('customer_two', 'user_two', 'password_two')
```

This will authenticate a second user. You can then switch between your open user sessions with *set_active_session* by passing a username. You can also pass the customer name as a keyword argument (in case you have the same username in two different customers).Use the *get_open_sessions* method to get a dictionary of all open sessions

```
>>> current_sessions = dynect_session.get_open_sessions()
>>> # loop through all open sessions
>>> for session in current_sessions:
...     dynect_session.set_active_session(session['user_name'], customer=session[
↪'customer'])
...     print("Zones for {0}".format(dynect_session.username))
...     print(get_all_zones())
```

log_out_active_session can be called to only log out of the active session. Calling *log_out* will log out of all open sessions

DynectSession as a Context Manager

As of version 1.2.0 you have the ability to use a DynectSession as a context manager, like so

```
>>> from dyn.tm.session import DynectSession
>>> with DynectSession('customer', 'user', 'password') as s:
...     return s.permissions
```

This feature is particularly useful if you're looking to manage multiple user accounts programatically.

Overriding Sessions

As of version 1.2.0 you have the ability to override an existing DynectSession with the use of the new_session class method like so

```
>>> from dyn.tm.session import DynectSession
>>> s = DynectSession('customer', 'user', 'password')
>>> s = DynectSession.new_session('customer', 'another_user', 'password')
```

Getting Sessions

If you don't want to track your current DynectSession, but want to be able to access your current one later, you can make use of the get_session class method like so

```
>>> from dyn.tm.session import DynectSession
>>> DynectSession('customer', 'user', 'password')
>>> DynectSession.get_session().username
'user'
```

Session History

As of version 1.3.0 users can now optionally allow DynectSessions to store a history of API calls that are made. This can be particularly useful for debugging, as well as for use when contacting Support.

```
>>> >>> from dyn.tm.session import DynectSession
>>> s = DynectSession('customer', 'user', 'password', history=True)
>>> s.history
... [('2014-10-14T11:15:17.351740',
...   '/REST/Session/',
...   'POST',
...   {'customer_name': 'customer', 'password': '*****', 'user_name': 'user'},
...   u'success')]
```

Please note that if you do not specify *history* as *True* when you log in, that your history will not be recorded and *s.history* will return *None*

Zones

The `zones` module contains interfaces for all of the various Zone management features offered by the dyn.tm REST API

List Functions

The following function is primarily a helper function which performs an API "Get All" call. This function returns a single `list` of `Zone` objects.

`dyn.tm.zones.`**`get_all_zones`**`()`
Accessor function to retrieve a *list* of all *`Zone`*'s accessible to a user

Returns a *list* of *`Zone`*'s

`dyn.tm.zones.`**`get_all_secondary_zones`**`()`
Accessor function to retrieve a *list* of all `SecondaryZone`'s accessible to a user

Returns a *list* of *`SecondaryZone`*'s

Classes

Zone

class `dyn.tm.zones.`**`Zone`**(*name*, **args*, ***kwargs*)
A class representing a DynECT Zone

`__init__`(*name*, **args*, ***kwargs*)

Create a `Zone` object. Note: When creating a new `Zone` if no contact is specified the path to a local zone file must be passed to the `file_name` param.

Parameters

- **name** – the name of the zone to create
- **contact** – Administrative contact for this zone
- **ttl** – TTL (in seconds) for records in the zone
- **serial_style** – The style of the zone's serial. Valid values: increment, epoch, day, minute
- **file_name** – The path to a valid RFC1035, BIND, or tinydns style Master file. Note: this file must be under 1mb in size.
- **master_ip** – The IP of the master server from which to fetch zone data for Transferring this `Zone`. Note: This argument is required for performing a valid ZoneTransfer operation.
- **timeout** – The time, in minutes, to wait for a zone xfer to complete

add_record(*name=None*, *record_type='A'*, **args*, ***kwargs*)
Adds an a record with the provided name and data to this `Zone`

Parameters

- **name** – The name of the node where this record will be added
- **record_type** – The type of record you would like to add. Valid record_type arguments are: 'A', 'AAAA', 'CERT', 'CNAME', 'DHCID', 'DNAME', 'DNSKEY', 'DS', 'KEY', 'KX', 'LOC', 'IPSECKEY', 'MX', 'NAPTR', 'PTR', 'PX', 'NSAP', 'RP', 'NS', 'SOA', 'SPF', 'SRV', and 'TXT'.
- **args** – Non-keyword arguments to pass to the Record constructor
- **kwargs** – Keyword arguments to pass to the Record constructor

add_service(*name=None*, *service_type=None*, **args*, ***kwargs*)
Add the specified service type to this zone, or to a node under this zone

Parameters

- **name** – The name of the `Node` where this service will be attached to or *None* to attach it to the root `Node` of this `Zone`
- **service_type** – The type of the service you would like to create. Valid service_type arguments are: 'ActiveFailover', 'DDNS', 'DNSSEC', 'DSF', 'GSLB', 'RDNS', 'RTTM', 'HTTPRedirect'
- **args** – Non-keyword arguments to pass to the Record constructor
- **kwargs** – Keyword arguments to pass to the Record constructor

contact
The email address of the primary `Contact` associated with this `Zone`

delete()
Delete this `Zone` and perform nessecary cleanups

fqdn
The name of this `Zone`

freeze()
Causes the zone to become frozen. Freezing a zone prevents changes to the zone until it is thawed.

get_all_active_failovers()
Retrieve a list of all ActiveFailover services associated with this Zone

Returns A List of ActiveFailover Services

get_all_advanced_redirect()
Retrieve a list of all AdvancedRedirect services associated with this Zone

Returns A List of AdvancedRedirect Services

get_all_ddns()
Retrieve a list of all DDNS services associated with this Zone

Returns A List of DDNS Services

get_all_gslb()
Retrieve a list of all GSLB services associated with this Zone

Returns A List of GSLB Services

get_all_httpredirect()
Retrieve a list of all HTTPRedirect services associated with this Zone

Returns A List of HTTPRedirect Services

get_all_nodes()
Returns a list of Node Objects for all subnodes in Zone (Excluding the Zone itself.)

get_all_rdns()
Retrieve a list of all ReverseDNS services associated with this Zone

Returns A List of ReverseDNS Services

get_all_records()
Retrieve a list of all record resources for the specified node and zone combination as well as all records from any Base_Record below that point on the zone hierarchy

Returns A List of all the DNSRecord's under this Zone

get_all_records_by_type(*record_type*)
Get a list of all DNSRecord of type record_type which are owned by this node.

Parameters **record_type** – The type of DNSRecord you wish returned. Valid record_type arguments are: 'A', 'AAAA', 'CAA', 'CERT', 'CNAME', 'DHCID', 'DNAME', 'DNSKEY', 'DS', 'KEY', 'KX', 'LOC', 'IPSECKEY', 'MX', 'NAPTR', 'PTR', 'PX', 'NSAP', 'RP', 'NS', 'SOA', 'SPF', 'SRV', and 'TXT'.

Returns A List of DNSRecord's

get_all_rttm()
Retrieve a list of all RTTM services associated with this Zone

Returns A List of RTTM Services

get_any_records()
Retrieve a list of all DNSRecord's associated with this Zone

get_node(*node=None*)
Returns all DNS Records for that particular node

Parameters **node** – The name of the Node you wish to access, or *None* to get the root Node of this Zone

get_notes(*offset=None*, *limit=None*)
Generates a report containing the Zone Notes for this Zone

Parameters

- **offset** – The starting point at which to retrieve the notes
- **limit** – The maximum number of notes to be retrieved

Returns A `list` of `dict` containing `Zone` Notes

get_qps(*start_ts*, *end_ts=None*, *breakdown=None*, *hosts=None*, *rrecs=None*)
Generates a report with information about Queries Per Second (QPS) for this zone

Parameters

- **start_ts** – datetime.datetime instance identifying point in time for the QPS report
- **end_ts** – datetime.datetime instance indicating the end of the data range for the report. Defaults to datetime.datetime.now()
- **breakdown** – By default, most data is aggregated together. Valid values ('hosts', 'rrecs', 'zones').
- **hosts** – List of hosts to include in the report.
- **rrecs** – List of record types to include in report.

Returns A `str` with CSV data

name
The name of this `Zone`

publish(*notes=None*)
Causes all pending changes to become part of the zone. The serial number increments based on its serial style and the data is pushed out to the nameservers.

serial
The current serial of this `Zone`

serial_style
The current serial style of this `Zone`

status
Convenience property for `Zones`. If a `Zones` is frozen the status will read as *'frozen'*, if the `Zones` is not frozen the status will read as *'active'*. Because the API does not return information about whether or not a `Zones` is frozen there will be a few cases where this status will be *None* in order to avoid guessing what the current status actually is.

task
`Task` for most recent system action on this `Zone`.

thaw()
Causes the zone to become thawed. Thawing a frozen zone allows changes to again be made to the zone.

ttl
This `Zone`'s default TTL

Zone Examples

The following examples highlight how to use the `Zone` class to get/create `Zone`'s on the dyn.tm System and how to edit these objects from within a Python script.

Creating a new Zone

The following example shows how to create a new `Zone` on the dyn.tm System and how to edit some of the fields using the returned `Zone` object.

```
>>> from dyn.tm.zones import Zone
>>> # Create a dyn.tmSession
>>> new_zone = Zone('myzone.com', 'me@email.com')
>>> new_zone.serial
0
>>> new_zone.publish()
>>> new_zone.serial
1
```

Getting an Existing Zone

The following example shows how to get an existing `Zone` from the dyn.tm System.

```
>>> from dyn.tm.zones import Zone
>>> # Create a dyn.tmSession
>>> my_zone = Zone('myzone.com')
>>> my_zone.serial
5
>>> my_zone.contact
u'myemail@email.com'
```

Using lists of Zones

The following example shows how to use the results of a call to the `get_all_zones()` functions

```
>>> from dyn.tm.zones import get_all_zones
>>> # Create a dyn.tmSession
>>> my_zones = get_all_zones()
>>> for zone in my_zones:
...     if zone.serial_style != 'increment':
...         zone.serial_style = 'increment'
```

Adding Records to a Zone

The following examples show how to add records to a Zone using the add_record method.

```
>>> from dyn.tm.zones import Zone
>>> # Create a dyn.tmSession
>>> my_zone = Zone('myzone.com')
# Add record to zone apex
>>> my_zone.add_record(record_type='MX', exchange='mail.example.com.')
# Add record to node under zone apex
>>> my_zone.add_record('my_node', record_type='A', address='1.1.1.1')
```

Secondary Zone

class dyn.tm.zones.**SecondaryZone**(*zone*, **args*, ***kwargs*)
A class representing DynECT Secondary zones

__init__(*zone*, **args*, ***kwargs*)
Create a SecondaryZone object

Parameters

- **zone** – The name of this secondary zone
- **masters** – A list of IPv4 or IPv6 addresses of the master nameserver(s) for this zone.
- **contact_nickname** – Name of the Contact that will receive notifications for this zone
- **tsig_key_name** – Name of the TSIG key that will be used to sign transfer requests to this zone's master

activate()
Activates this secondary zone

active
Reports the status of SecondaryZone Y, L or N

contact_nickname
Name of the Contact that will receive notifications for this zone

deactivate()
Deactivates this secondary zone

delete()
Delete this SecondaryZone

masters
A list of IPv4 or IPv6 addresses of the master nameserver(s) for this zone.

retransfer()
Retransfers this secondary zone from its original provider into Dyn's Managed DNS

serial
Reports the serial of SecondaryZone

task
Task for most recent system action on this SecondaryZone.

tsig_key_name
Name of the TSIG key that will be used to sign transfer requests to this zone's master

zone
The name of this SecondaryZone

Secondary Zone Examples

The following examples highlight how to use the SecondaryZone class to get/create SecondaryZone's on the dyn.tm System and how to edit these objects from within a Python script.

Creating a new Secondary Zone

The following example shows how to create a new `SecondaryZone` on the dyn.tm System and how to edit some of the fields using the returned `SecondaryZone` object.

```
>>> from dyn.tm.zones import SecondaryZone
>>> # Create a dyn.tmSession
>>> new_zone = SecondaryZone('myzone.com', '127.0.0.1', 'mynickame')
>>> new_zone.active
'Y'
>>> new_zone.retransfer()
>>> new_zone.serial
1
```

Getting an Existing Secondary Zone

The following example shows how to get an existing `SecondaryZone` from the dyn.tm System.

```
>>> from dyn.tm.zones import SecondaryZone
>>> # Create a dyn.tmSession
>>> my_zone = SecondaryZone('myzone.com')
>>> my_zone.serial
5
>>> my_zone.contact_nickname
u'mynickname'
>>> my_zone.deactivate()
>>> my_zone.active
'N'
```

Using lists of Secondary Zones

The following example shows how to use the results of a call to the `get_all_secondary_zones()` functions

```
>>> from dyn.tm.zones import get_all_secondary_zones
>>> my_sec_zones = get_all_secondary_zones()
>>> for zone in my_sec_zones:
...     zone.activate()
```

Node

It is important to note that creation of a Node class will not immediately take affect on the dyn.tm System unless it is created via a `Zone` instance. While creating `Node`'s via a `Zone` you are required to place either a `DNSRecord` or a service on that `Node` which allows it to be created. To clerify, because `Node`'s may not exist without either a record or service `node = Node('zone.com', 'fqdn.zone.com.')` will not actually create anything on the Dyn side until you add a record or service, whereas `rec = zone.add_record('fqnd', 'A', '127.0.0.1')` will create a new `Node` named 'fqdn' with an `ARecord` attached.

class `dyn.tm.zones.`**Node**(*zone*, *fqdn=None*)

Node object. Represents a valid fqdn node within a zone. It should be noted that simply creating a `Node` object does not actually create anything on the DynECT System. The only way to actively create a `Node` on the DynECT System is by attaching either a record or a service to it.

__init__(*zone*, *fqdn=None*)
Create a `Node` object

Parameters

- **zone** – name of the zone that this Node belongs to
- **fqdn** – the fully qualified domain name of this zone

add_record(*record_type='A'*, **args*, ***kwargs*)
Adds an a record with the provided data to this `Node`

Parameters

- **record_type** – The type of record you would like to add. Valid record_type arguments are: 'A', 'AAAA', 'CERT', 'CNAME', 'DHCID', 'DNAME', 'DNSKEY', 'DS', 'KEY', 'KX', 'LOC', 'IPSECKEY', 'MX', 'NAPTR', 'PTR', 'PX', 'NSAP', 'RP', 'NS', 'SOA', 'SPF', 'SRV', and 'TXT'.
- **args** – Non-keyword arguments to pass to the Record constructor
- **kwargs** – Keyword arguments to pass to the Record constructor

add_service(*service_type=None*, **args*, ***kwargs*)
Add the specified service type to this `Node`

Parameters

- **service_type** – The type of the service you would like to create. Valid service_type arguments are: 'ActiveFailover', 'DDNS', 'DNSSEC', 'DSF', 'GSLB', 'RDNS', 'RTTM', 'HTTPRedirect'
- **args** – Non-keyword arguments to pass to the Record constructor
- **kwargs** – Keyword arguments to pass to the Record constructor

delete()
Delete this node, any records within this node, and any nodes underneath this node

get_all_records()
Retrieve a list of all record resources for the specified node and zone combination as well as all records from any Base_Record below that point on the zone hierarchy

get_all_records_by_type(*record_type*)
Get a list of all `DNSRecord` of type `record_type` which are owned by this node.

Parameters **record_type** – The type of `DNSRecord` you wish returned. Valid record_type arguments are: 'A', 'AAAA', 'CERT', 'CNAME', 'DHCID', 'DNAME', 'DNSKEY', 'DS', 'KEY', 'KX', 'LOC', 'IPSECKEY', 'MX', 'NAPTR', 'PTR', 'PX', 'NSAP', 'RP', 'NS', 'SOA', 'SPF', 'SRV', and 'TXT'.

Returns A list of `DNSRecord`'s

get_any_records()
Retrieve a list of all recs

Node Examples

The following examples highlight how to use the `Node` class to get/create `Zone`'s on the dyn.tm System and how to edit these objects from within a Python script.

Creating a new Node

The following example shows how to create a new `Node` on the dyn.tm System and how to edit some of the fields using the returned `Node` object. The easiest way to manipulate `Node` objects is via a `Zone` object. This example will show how to create new `Node` objects both by using a `Zone` as a proxy, and by creating a `Node` as a standalone object.

```
>>> from dyn.tm.zones import Zone
>>> # Create a dyn.tmSession
>>> new_zone = Zone('myzone.com', 'me@email.com')
>>> new_zone.add_record('NewNode', 'A', '127.0.0.1')
<ARecord>: 127.0.0.1
>>> node = new_zone.get_node('NewNode')
>>> node.add_record('A', '127.0.1.1')
<ARecord>: 127.0.1.1
>>> node.get_any_records()
{u'a_records': [<ARecord>: 127.0.0.1, <ARecord>: 127.0.1.1]}
```

```
>>> from dyn.tm.zones import Node
>>> # Create a dyn.tmSession
>>> # Assuming the :class:`Zone` from the above example still exists
>>> new_node = Node('myzone.com', 'NewNode.myzone.com')
>>> new_node.get_any_records()
{u'a_records': [<ARecord>: 127.0.0.1, <ARecord>: 127.0.1.1]}
```

Getting an Existing Node

The following example shows how to get an existing `Node` from the dyn.tm System. Similarly to the above examples the easiest way to manipulate existing `Node` objects is via a `Zone` object.

```
>>> from dyn.tm.zones import Zone
>>> # Create a dyn.tmSession
>>> new_zone = Zone('myzone.com', 'me@email.com')
>>> new_zone.add_record('NewNode', 'A', '127.0.0.1')
>>> node = new_zone.get_node('NewNode')
>>> node.get_any_records()
{u'a_records': ['127.0.0.1'], ...}
```

```
>>> from dyn.tm.zones import Node
>>> # Create a dyn.tmSession
>>> # Assuming the :class:`Zone` from the above example still exists
>>> new_node = Node('myzone.com', 'NewNode.myzone.com')
>>> new_node.get_any_records()
{u'a_records': ['127.0.0.1'], ...}
```

Accounts

The `accounts` module contains interfaces for all of the various Account management features offered by the dyn.tm REST API

Search/List Functions

The following functions are primarily helper functions which will perform API "Get All" calls. These functions all return a single `list` containing class representations of their respective types. For instance `get_all_users()` returns a `list` of `User` objects.

`dyn.tm.accounts.`**`get_updateusers`**(*search=None*)
Return a `list` of *`UpdateUser`* objects. If *search* is specified, then only `UpdateUsers` who match those search criteria will be returned in the list. Otherwise, all `UpdateUsers`'s will be returned.

Parameters **`search`** – A `dict` of search criteria. Key's in this `dict` much map to an attribute a `UpdateUsers` instance and the value mapped to by that key will be used as the search criteria for that key when searching.

Returns a `list` of *`UpdateUser`* objects

`dyn.tm.accounts.`**`get_users`**(*search=None*)
Return a `list` of *`User`* objects. If *search* is specified, then only users who match those search parameters will be returned in the list. Otherwise, all *`User`*'s will be returned.

Parameters **`search`** – A `dict` of search criteria. Key's in this `dict` much map to an attribute a *`User`* instance and the value mapped to by that key will be used as the search criteria for that key when searching.

Returns a `list` of *`User`* objects

`dyn.tm.accounts.`**`get_permissions_groups`**(*search=None*)
Return a `list` of `PermissionGroup` objects. If *search* is specified, then only `PermissionGroup`'s that match those search criteria will be returned in the list. Otherwise, all `PermissionGroup`'s will be returned.

Parameters **`search`** – A `dict` of search criteria. Key's in this `dict` much map to an attribute a `PermissionGroup` instance and the value mapped to by that key will be used as the search criteria for that key when searching.

Returns a `list` of `PermissionGroup` objects

`dyn.tm.accounts.`**`get_contacts`**(*search=None*)
Return a `list` of *`Contact`* objects. If *search* is specified, then only *`Contact`*'s who match those search criteria will be returned in the list. Otherwise, all *`Contact`*'s will be returned.

Parameters **`search`** – A `dict` of search criteria. Key's in this `dict` much map to an attribute a *`Contact`* instance and the value mapped to by that key will be used as the search criteria for that key when searching.

Returns a `list` of *`Contact`* objects

`dyn.tm.accounts.`**`get_notifiers`**(*search=None*)
Return a `list` of *`Notifier`* objects. If *search* is specified, then only *`Notifier`*'s who match those search criteria will be returned in the list. Otherwise, all *`Notifier`*'s will be returned.

Parameters **`search`** – A `dict` of search criteria. Key's in this `dict` much map to an attribute a *`Notifier`* instance and the value mapped to by that key will be used as the search criteria for that key when searching.

Returns a `list` of *`Notifier`* objects

Search/List Function Examples

Using these search functions is a fairly straightforward endeavour, you can either leave your search criteria as None and get a list of ALL objects of that type, or you can specify a search dict like so

```
>>> from dyn.tm.accounts import get_users
>>> all_users = get_users()
>>> all_users
[User: <jnappi>, User: <rshort>, User: <tmpuser35932>,...]
>>> search_criteria = {'first_name': 'Jon'}
>>> jons = get_users(search_criteria)
>>> jons
[User: <jnappi>, User: <jsmith>]
```

Classes

UpdateUser

class dyn.tm.accounts.**UpdateUser**(**args*, ***kwargs*)

UpdateUser type objects are a special form of a *User* which are tied to a specific Dynamic DNS services.

__init__(**args*, ***kwargs*)

Create an *UpdateUser* object

Parameters

- **user_name** – the Username this *UpdateUser* uses or will use to log in to the DynECT System. A *UpdateUser*'s *user_name* is required for both creating and getting *UpdateUser*'s.
- **nickname** – When creating a new *UpdateUser* on the DynECT System, this *nickname* will be the System nickname for this *UpdateUser*
- **password** – When creating a new *UpdateUser* on the DynECT System, this *password* will be the password this *UpdateUser* uses to log into the System

block()

Set the status of this *UpdateUser* to 'blocked'. This will prevent this *UpdateUser* from logging in until they are explicitly unblocked.

delete()

Delete this *UpdateUser* from the DynECT System. It is important to note that this operation may not be undone.

nickname

This UpdateUser`s `nickname. An *UpdateUser*'s *nickname* is a read-only property which can not be updated after the *UpdateUser* has been created.

password

The current *password* for this *UpdateUser*. An *UpdateUser*'s *password* may be reassigned.

status

The current *status* of an *UpdateUser* will be one of either 'active' or 'blocked'. Blocked *UpdateUser*'s are unable to log into the DynECT System, where active *UpdateUser*'s are.

sync_password()

Pull in this *UpdateUser* current password from the DynECT System, in the unlikely event that this *UpdateUser* object's password may have gotten out of sync

unblock()

Set the status of this *UpdateUser* to 'active'. This will re-enable this *UpdateUser* to be able to login if they were previously blocked.

user_name
This *UpdateUser*'s *user_name*. An *UpdateUser*'s user_name is a read-only property which can not be updated after the UpdateUser has been created.

UpdateUser Examples

The following examples highlight how to use the UpdateUser class to get/create UpdateUser's on the dyn.tm System and how to edit these objects from within a Python script.

Creating a new UpdateUser

The following example shows how to create a new UpdateUser on the dyn.tm System and how to edit some of the fields using the returned UpdateUser object.

```
>>> from dyn.tm.accounts import UpdateUser
>>> # Create a dyn.tmSession
>>> new_user = UpdateUser('ausername', 'anickname', 'passw0rd')
>>> new_user.user_name
u'ausername'
>>> newuser.nickname
u'anickname'
>>> new_user.block()
>>> new_user.status
u'blocked'
>>> new_user.unblock()
>>> new_user.password = 'anewpassword'
>>> new_user.password
u'anewpassword'
```

Getting an Existing UpdateUser

The following example shows how to get an existing UpdateUser from the dyn.tm System and how to edit some of the same fields mentioned above.

```
>>> from dyn.tm.accounts import UpdateUser
>>> # Create a dyn.tmSession
>>> my_user = UpdateUser('myusername')
>>> my_user.user_name
u'myusername'
>>> my_user.status
u'blocked'
>>> my_user.unblock()
>>> my_user.status
u'active'
```

User

class dyn.tm.accounts.**User**(*user_name*, **args*, ***kwargs*)
DynECT System User object

__init__(*user_name*, **args*, ***kwargs*)
Create a new *User* object

Parameters

- **user_name** – This *User*'s system username; used for logging into the system
- **password** – Password for this *User* account
- **email** – This *User*'s Email address
- **first_name** – This *User*'s first name
- **last_name** – This *User*'s last name
- **nickname** – The nickname for the *Contact* associated with this *User*
- **organization** – This *User*'s organization
- **phone** – This *User*'s phone number. Can be of the form: (0) (country-code) (local number) (extension) Only the country-code (1-3 digits) and local number (at least 7 digits) are required. The extension can be up to 4 digits. Any non-digits are ignored.
- **address** – This *User*'s street address
- **address2** – This *User*'s street address, line 2
- **city** – This *User*'s city, part of the user's address
- **country** – This *User*'s country, part of the user's address
- **fax** – This *User*'s fax number
- **notify_email** – Email address where this *User* should receive notifications
- **pager_email** – Email address where this *User* should receive messages destined for a pager
- **post_code** – Zip code or Postal code
- **group_name** – A list of permission groups this *User* belongs to
- **permission** – A list of permissions assigned to this *User*
- **zone** – A list of zones where this *User*'s permissions apply
- **forbid** – A list of forbidden permissions for this *User*
- **status** – Current status of this *User*
- **website** – This *User*'s website

add_forbid_rule (*permission*, *zone=None*)
Adds the forbid rule to the *User*'s permission group

Parameters

- **permission** – the permission to forbid from this *User*
- **zone** – A list of zones where the forbid rule applies

add_permission (*permission*)
Add individual permissions to this *User*

Parameters **permission** – the permission to add

add_permissions_group (*group*)
Assigns the permissions group to this *User*

Parameters **group** – the permissions group to add to this *User*

add_zone(*zone*, *recurse='Y'*)
Add individual zones to this *User* :param zone: the zone to add :param recurse: determine if permissions should be extended to

subzones.

address
This *User*'s street address

address_2
This *User*'s street address, line 2

block()
Blocks this *User* from logging in

city
This *User*'s city, part of the user's address

country
This *User*'s country, part of the user's address

delete()
Delete this *User* from the system

delete_forbid_rule(*permission*, *zone=None*)
Removes a forbid permissions rule from the *User*'s permission group

Parameters

- **permission** – permission
- **zone** – A list of zones where the forbid rule applies

delete_permission(*permission*)
Remove this specific permission from the *User*

Parameters **permission** – the permission to remove

delete_permissions_group(*group*)
Removes the permissions group from the *User*

Parameters **group** – the permissions group to remove from this *User*

delete_zone(*zone*)
Remove this specific zones from the *User*

Parameters **zone** – the zone to remove

email
This *User*'s Email address

fax
This *User*'s fax number

first_name
This *User*'s first name

forbid
A list of forbidden permissions for this *User*

group_name
A list of permission groups this *User* belongs to

last_name
This *User*'s last name

nickname
The nickname for the *Contact* associated with this *User*

notify_email
Email address where this *User* should receive notifications

organization
This *User*'s organization

pager_email
Email address where this *User* should receive messages destined for a pager

permission
A list of permissions assigned to this *User*

phone
This *User*'s phone number. Can be of the form: (0) (country-code) (local number) (extension) Only the country-code (1-3 digits) and local number (at least 7 digits) are required. The extension can be up to 4 digits. Any non-digits are ignored.

post_code
This *User*'s postal code, part of the user's address

replace_forbid_rules (*forbid=None*)
Replaces the list of forbidden permissions in the *User*'s permissions group with a new list.

Parameters **forbid** – A list of rules to replace the forbidden rules on the *User*'s permission group. If empty or not passed in, the *User*'s forbid list will be cleared

replace_permission (*permission=None*)
Replaces the list of permissions for this *User*

Parameters **permissions** – A list of permissions. Pass an empty list or omit the argument to clear the list of permissions of the *User*

replace_permissions_group (*groups=None*)
Replaces the list of permissions for this *User*

Parameters **groups** – A list of permissions groups. Pass an empty list or omit the argument to clear the list of permissions groups of the *User*

replace_zones (*zones*)
Remove this specific zones from the *User* :param zones: array of the zones to be updated format must be [{'zone_name':[yourzone], recurse: 'Y'},{ ...}] recurse is optional.

status
A *User*'s status is a read-only property. To change you must use the `block()`/`unblock()` methods

unblock()
Restores this *User* to an active status and re-enables their log-in

user_name
A *User*'s user_name is a read-only property

website
This *User*'s website

zone
A list of zones where this *User*'s permissions apply

User Examples

The following examples highlight how to use the `User` class to get/create `User`'s on the dyn.tm System and how to edit these objects from within a Python script.

Creating a new User

The following example shows how to create a new `User` on the dyn.tm System and how to edit some of the fields using the returned `User` object.

```
>>> from dyn.tm.accounts import User
>>> # Create a dyn.tmSession
>>> new_user = User('newuser', 'passw0rd', 'contact@email.com', 'first',
...                 'last', 'nickname', 'MyOrganization', '(123)456-7890')
>>> new_user.status
u'active'
>>> new_user.city
None
>>> new_user.city = 'Manchester'
>>> new_user.permission
['ZoneGet', 'ZoneUpdate']
>>> new_user.add_permission('ZoneCreate')
>>> new_user.permission
['ZoneGet', 'ZoneUpdate', 'ZoneCreate']
```

Getting an Existing User

The following example shows how to get an existing `User` from the dyn.tm System and how to edit some of the same fields mentioned above.

```
>>> from dyn.tm.accounts import User
>>> # Create a dyn.tmSession
>>> my_user = User('myusername')
>>> my_user.status
u'blocked'
>>> my_user.unblock()
>>> my_user.status
u'active'
```

PermissionsGroup

class dyn.tm.accounts.**PermissionsGroup**(*group_name*, **args*, ***kwargs*)

A DynECT System Permissions Group object

__init__(*group_name*, **args*, ***kwargs*)

Create a new permissions Group

Parameters

- **group_name** – The name of the permission group to update
- **description** – A description of the permission group
- **group_type** – The type of the permission group. Valid values: plain or default

- **all_users** – If 'Y', all current users will be added to the group. Cannot be used if user_name is passed in
- **permission** – A list of permissions that the group contains
- **user_name** – A list of users that belong to the permission group
- **subgroup** – A list of groups that belong to the permission group
- **zone** – A list of zones where the group's permissions apply

add_permission(*permission*)
Adds individual permissions to the user

Parameters **permission** – the permission to add to this user

add_subgroup(*name*)
Add a new Sub group to this *PermissionsGroup*

Parameters **name** – The name of the *PermissionsGroup* to be added to this *PermissionsGroup*'s subgroups

add_zone(*zone*, *recurse='Y'*)
Add a new Zone to this *PermissionsGroup*

Parameters

- **zone** – The name of the Zone to be added to this *PermissionsGroup*
- **recurse** – A flag determining whether or not to add all sub-nodes of a Zone to this *PermissionsGroup*

all_users
If 'Y', all current users will be added to the group. Cannot be used if user_name is passed in

delete()
Delete this permission group

delete_subgroup(*name*)
Remove a Subgroup from this *PermissionsGroup*

Parameters **name** – The name of the *PermissionsGroup* to be remoevd from this *PermissionsGroup*'s subgroups

description
A description of this permission group

group_name
The name of this permission group

group_type
The type of this permission group

permission
A list of permissions that this group contains

remove_permission(*permission*)
Removes the specific permission from the user

Parameters **permission** – the permission to remove

replace_permissions(*permission=None*)
Replaces a list of individual user permissions for the user

Parameters **permission** – A list of permissions. Pass an empty list or omit the argument to clear the list of permissions of the user

subgroup
A list of groups that belong to the permission group

update_subgroup (*subgroups*)
Update the subgroups under this *PermissionsGroup*

Parameters **subgroups** – The subgroups with updated information

user_name
A list of users that belong to the permission group

zone
A list of users that belong to the permission group

PermissionsGroup Examples

The following examples highlight how to use the `PermissionsGroup` class to get/create `PermissionsGroup`'s on the dyn.tm System and how to edit these objects from within a Python script.

Creating a new PermissionsGroup

The following example shows how to create a new `PermissionsGroup` on the dyn.tm System and how to edit some of the fields using the returned `PermissionsGroup` object.

```
>>> from dyn.tm.accounts import PermissionsGroup
>>> # Create a dyn.tmSession
>>> new_group = PermissionsGroup('newgroupname', 'description_of_new_group')
>>> new_group.type
u'default'
>>> new_group.add_permission('ZoneUpdate')
>>> new_group.permission
['ZoneUpdate']
>>> # Note that assigning new_group.permission will clear all permissions
>>> new_group.permission = ['ZoneGet']
>>> new_group.permission
['ZoneGet']
>>> # Also note this is functionally equivalent to calling replace_permission
>>> new_group.replace_permission(['ZoneCreate'])
>>> new_group.permission
['ZoneCreate']
```

Getting an Existing PermissionsGroup

The following example shows how to get an existing `User` from the dyn.tm System and how to edit some of the same fields mentioned above.

```
>>> from dyn.tm.accounts import PermissionsGroup
>>> # Create a dyn.tmSession
>>> my_group = PermissionsGroup('newgroupname')
>>> my_group.type
u'default'
>>> my_group.type = 'plain'
>>> my_group.type
u'plain'
>>> my_group.description = 'A better group description.'
```

```
>>> my_group.description
u'A better group description.'
```

Notifier

class dyn.tm.accounts.**Notifier**(**args*, ***kwargs*)

DynECT System Notifier

__init__(**args*, ***kwargs*)

Create a new *Notifier* object

Parameters

- **label** – The label used to identify this *Notifier*
- **recipients** – List of Recipients attached to this *Notifier*
- **services** – List of services attached to this *Notifier*
- **notifier_id** – The system id of this *Notifier*

delete()

Delete this *Notifier* from the Dynect System

label

The label used to identify this *Notifier*

notifier_id

The unique System id for this Notifier

recipients

List of Recipients attached to this *Notifier*

services

List of services attached to this *Notifier*

Notifier Examples

The following examples highlight how to use the `Notifier` class to get/create `Notifier`'s on the dyn.tm System and how to edit these objects from within a Python script.

Creating a new Notifier

The following example shows how to create a new `Notifier` on the dyn.tm System and how to edit some of the fields using the returned `Notifier` object.

```
>>> from dyn.tm.accounts import Notifier
>>> # Create a dyn.tmSession
>>> new_notif = Notifier(label='notifierlabel')
>>> new_notif.services
[]
>>> new_notif.recipients
[]
>>> # Probably want to include more
```

Getting an Existing Notifier

The following example shows how to get an existing `Notifier` from the dyn.tm System and how to edit some of the same fields mentioned above.

```
>>> from dyn.tm.accounts import Notifier
>>> # Create a dyn.tmSession
>>> # Note that in order to get a Notifier you will need the ID of that Notifier
>>> my_notif = Notifier(my_notifier_id)
>>> my_notif.services
[]
>>> my_notif.recipients
[]
>>> # Probably want to include more
```

Contact

class dyn.tm.accounts.**Contact**(*nickname*, **args*, ***kwargs*)

A DynECT System Contact

__init__(*nickname*, **args*, ***kwargs*)

Create a *Contact* object

Parameters

- **nickname** – The nickname for this *Contact*
- **email** – The *Contact*'s email address
- **first_name** – The *Contact*'s first name
- **last_name** – The *Contact*'s last name
- **organization** – The *Contact*'s organization
- **phone** – The *Contact*'s phone number. Can be of the form: (0) (country-code) (local number) (extension) Only the country-code (1-3 digits) and local number (at least 7 digits) are required. The extension can be up to 4 digits. Any non-digits are ignored.
- **address** – The *Contact*'s street address
- **address2** – The *Contact*'s street address, line 2
- **city** – The *Contact*'s city, part of the user's address
- **country** – The *Contact*'s country, part of the *Contact*'s address
- **fax** – The *Contact*'s fax number
- **notify_email** – Email address where the *Contact* should receive notifications
- **pager_email** – Email address where the *Contact* should receive messages destined for a pager
- **post_code** – Zip code or Postal code
- **state** – The *Contact*'s state, part of the *Contact*'s address
- **website** – The *Contact*'s website

address

This *Contact*'s street address

address_2
This *Contact*'s street address, line 2

city
This *Contact*'s city

country
This *Contact*'s Country

delete()
Delete this *Contact* from the Dynect System

email
This *Contact*'s DynECT System Email address

fax
The fax number associated with this *Contact*

first_name
The first name of this *Contact*

last_name
The last name of this *Contact*

nickname
This *Contact*'s DynECT System Nickname

notify_email
Email address where this *Contact* should receive notifications

organization
The organization this *Contact* belongs to within the DynECT System

pager_email
Email address where this *Contact* should receive messages destined for a pager

phone
The phone number associated with this *Contact*

post_code
This `Contacts`'s postal code, part of the contacts's address

state
This *Contact*'s state

website
This *Contact*'s website

Contact Examples

The following examples highlight how to use the `Contact` class to get/create `Contact`'s on the dyn.tm System and how to edit these objects from within a Python script.

Creating a new Contact

The following example shows how to create a new `Contact` on the dyn.tm System and how to edit some of the fields using the returned `Contact` object.

```
>>> from dyn.tm.accounts import Contact
>>> # Create a dyn.tmSession
>>> new_contact = Contact('mynickname', 'me@email.com', 'firstname',
...                       'lastname', 'MyOrganization')
>>> new_contact.city
None
>>> new_contact.city = 'Manchester'
>>> new_contact.city
u'Manchester'
```

Getting an Existing Contact

The following example shows how to get an existing `Contact` from the dyn.tm System and how to edit some of the same fields mentioned above. It is also probably worth mentioning that when a `User` is created a `Contact` is also created and is associated with that `User`. However, when a `User` is deleted the associated `Contact` is not deleted along with it, as it may still be associated with active services.

```
>>> from dyn.tm.accounts import Contact
>>> # Create a dyn.tmSession
>>> my_contact = Contact('mynickname')
>>> my_contact.email
u'me@email.com'
>>> my_contact.email = 'mynewemail@email.com'
>>> my_contact.email
u'mynewemail@email.com'
```

Records

The `records` module contains interfaces to all of the various DNS Record management features offered by the dyn.tm REST API

DNSRecord

The DNSRecord object serves as the parent class for all other record type objects within the `records` module. It provides shared functionality to the various record types and normally will not need to be used directly.

class `dyn.tm.records.`**DNSRecord**(*zone*, *fqdn*, *create=True*)

Base record object contains functionality to be used across all other record type objects

__init__(*zone*, *fqdn*, *create=True*)

delete()

Delete the current record

fqdn

Once the fqdn is set, it will be a read only property

geo_node

geo_rdata

rdata()

Return a records rdata

rec_name

record_id
The unique ID of this record from the DynECT System

ttl
The TTL for this record

zone
Once the zone is set, it will be a read only property

ARecord

class dyn.tm.records.**ARecord**(*zone*, *fqdn*, **args*, ***kwargs*)
The IPv4 Address (A) Record forward maps a host name to an IPv4 address.

__init__(*zone*, *fqdn*, **args*, ***kwargs*)
Create an *ARecord* object

Parameters

- **zone** – Name of zone where the record will be added
- **fqdn** – Name of node where the record will be added
- **address** – IPv4 address for the record
- **ttl** – TTL for this record

address
Return the value of this record's address property

rdata()
Return this *ARecord*'s rdata as a JSON blob

AAAARecord

class dyn.tm.records.**AAAARecord**(*zone*, *fqdn*, **args*, ***kwargs*)
The IPv6 Address (AAAA) Record is used to forward map hosts to IPv6 addresses and is the current IETF recommendation for this purpose.

__init__(*zone*, *fqdn*, **args*, ***kwargs*)
Create an *AAAARecord* object

Parameters

- **zone** – Name of zone where the record will be added
- **fqdn** – Name of node where the record will be added
- **address** – IPv6 address for the record
- **ttl** – TTL for this record

address
Return the value of this record's address property

rdata()
Return this *AAAARecord*'s rdata as a JSON blob

ALIASRecord

class dyn.tm.records.**ALIASRecord**(*zone*, *fqdn*, **args*, ***kwargs*)
The ALIAS Records map an alias (CNAME) to the real or canonical name that may lie inside or outside the current zone.

__init__(*zone*, *fqdn*, **args*, ***kwargs*)
Create an *ALIASRecord* object

Parameters

- **zone** – Name of zone where the record will be added
- **fqdn** – Name of node where the record will be added
- **alias** – Hostname
- **ttl** – TTL for this record

alias
Hostname

rdata()
Return this *ALIASRecord*'s rdata as a JSON blob

CAARecord

class dyn.tm.records.**CAARecord**(*zone*, *fqdn*, **args*, ***kwargs*)
Certification Authority Authorization (CAA) Resource Record

This record allows a DNS domain name holder to specify one or more Certification Authorities (CAs) authorized to issue certificates for that domain. CAA Resource Records allow a public Certification Authority to implement additional controls to reduce the risk of unintended certificate mis-issue. This document defines the syntax of the CAA record and rules for processing CAA records by certificate issuers.

see: https://tools.ietf.org/html/rfc6844

__init__(*zone*, *fqdn*, **args*, ***kwargs*)
Create a *CAARecord* object

Parameters

- **zone** – Name of zone where the record will be added
- **fqdn** – Name of node where the record will be added
- **flags** – A byte
- **tag** – A string defining the tag component of the <tag>=<value> record property. May be one of: issue: The issue property entry authorizes the holder of

  the domain name <Issuer Domain Name> or a party acting under the explicit authority of the holder of that domain name to issue certificates for the domain in which the property is published.

  issuewild: The issuewild property entry authorizes the holder of the domain name <Issuer Domain Name> or a party acting under the explicit authority of the holder of that domain name to issue wildcard certificates for the domain in which the property is published.

iodef: Specifies a URL to which an issuer MAY report certificate issue requests that are inconsistent with the issuer's Certification Practices or Certificate Policy, or that a Certificate Evaluator may use to report observation of a possible policy violation.

param value A string representing the value component of the property. This will be an issuer domain name or a URL.

param ttl TTL for this record. Use 0 for zone default

flags

rdata()

tag

value

CERTRecord

class dyn.tm.records.**CERTRecord**(*zone*, *fqdn*, **args*, ***kwargs*)

The Certificate (CERT) Record may be used to store either public key certificates or Certificate Revocation Lists (CRL) in the zone file.

__init__(*zone*, *fqdn*, **args*, ***kwargs*)

Create a *CERTRecord* object

Parameters

- **zone** – Name of zone where the record will be added
- **fqdn** – Name of node where the record will be added
- **format** – Numeric value for the certificate type
- **tag** – Numeric value for the public key certificate
- **algorithm** – Public key algorithm number used to generate the certificate
- **certificate** – The public key certificate
- **ttl** – TTL for this record in seconds

algorithm

Public key algorithm number used to generate the certificate

certificate

The public key certificate

format

Numeric value for the certificate type.

rdata()

Return this *CERTRecord*'s rdata as a JSON blob

tag

Numeric value for the public key certificate

CDNSKEYRecord

class dyn.tm.records.**CDNSKEYRecord**(*zone*, *fqdn*, **args*, ***kwargs*)

The CDNSKEY Record, or "Child DNSKEY", describes the public key of a public key (asymmetric) cryptographic algorithm used with DNSSEC.nis. This is the DNSKEY for a Child Zone

__init__(*zone*, *fqdn*, **args*, ***kwargs*)

Create a *DNSKEYRecord* object

Parameters

- **zone** – Name of zone where the record will be added
- **fqdn** – Name of node where the record will be added
- **protocol** – Numeric value for protocol
- **public_key** – The public key for the DNSSEC signed zone
- **algorithm** – Numeric value representing the public key encryption algorithm which will sign the zone. Must be one of 1 (RSA-MD5), 2 (Diffie-Hellman), 3 (DSA/SHA-1), 4 (Elliptic Curve), or 5 (RSA-SHA-1)
- **flags** – Numeric value confirming this is the zone's DNSKEY
- **ttl** – TTL for this record. Use 0 for zone default

algorithm

Public key encryption algorithm will sign the zone

flags

Numeric value confirming this is the zone's DNSKEY

protocol

Numeric value for protocol. Set to 3 for DNSSEC

public_key

The public key for the DNSSEC signed zone

rdata()

Return this *DNSKEYRecord*'s rdata as a JSON blob

CSYNCRecord

class dyn.tm.records.**CSYNCRecord**(*zone*, *fqdn*, **args*, ***kwargs*)

The CSYNC RRType contains, in its RDATA component, these parts: an SOA serial number, a set of flags, and a simple bit-list indicating the DNS RRTypes in the child that should be processed by the parental agent in order to modify the DNS delegation records within the parent's zone for the child DNS operator.

__init__(*zone*, *fqdn*, **args*, ***kwargs*)

Create a *DSRecord* object

Parameters

- **zone** – Name of zone where the record will be added
- **fqdn** – Name of node where the record will be added
- **soa_serial** – SOA serial to bind to this record.
- **flags** – list of flags ('soaminimum', 'immediate')
- **rectypes** – list of record types to bind to this record.

- **ttl** – TTL for this record

flags
The flags, in list form

rdata()
Return this *DSRecord*'s rdata as a JSON blob

rectypes
list of record types

soa_serial
SOA Serial

CDSRecord

class dyn.tm.records.**CDSRecord**(*zone*, *fqdn*, **args*, ***kwargs*)
The Child Delegation Signer (CDS) record type is used in DNSSEC to create the chain of trust or authority from a signed child zone to a signed parent zone.

__init__(*zone*, *fqdn*, **args*, ***kwargs*)
Create a *DSRecord* object

Parameters

- **zone** – Name of zone where the record will be added
- **fqdn** – Name of node where the record will be added
- **digest** – The digest in hexadecimal form. 20-byte, hexadecimal-encoded, one-way hash of the DNSKEY record surrounded by parenthesis characters '(' & ')'
- **keytag** – The digest mechanism to use to verify the digest
- **algorithm** – Numeric value representing the public key encryption algorithm which will sign the zone. Must be one of 1 (RSA-MD5), 2 (Diffie-Hellman), 3 (DSA/SHA-1), 4 (Elliptic Curve), or 5 (RSA-SHA-1)
- **digtype** – the digest mechanism to use to verify the digest. Valid values are SHA1, SHA256
- **ttl** – TTL for this record. Use 0 for zone default

algorithm
Identifies the encoding algorithm

digest
The digest in hexadecimal form. 20-byte, hexadecimal-encoded, one-way hash of the DNSKEY record surrounded by parenthesis characters

digtype
Identifies which digest mechanism to use to verify the digest

keytag
Identifies which digest mechanism to use to verify the digest

rdata()
Return this *DSRecord*'s rdata as a JSON blob

CNAMERecord

class dyn.tm.records.**CNAMERecord**(*zone*, *fqdn*, **args*, ***kwargs*)

The Canonical Name (CNAME) Records map an alias to the real or canonical name that may lie inside or outside the current zone.

__init__(*zone*, *fqdn*, **args*, ***kwargs*)

Create an *CNAMERecord* object

Parameters

- **zone** – Name of zone where the record will be added
- **fqdn** – Name of node where the record will be added
- **cname** – Hostname
- **ttl** – TTL for this record

cname

Hostname

rdata()

Return this *CNAMERecord*'s rdata as a JSON blob

DHCIDRecord

class dyn.tm.records.**DHCIDRecord**(*zone*, *fqdn*, **args*, ***kwargs*)

The *DHCIDRecord* provides a means by which DHCP clients or servers can associate a DHCP client's identity with a DNS name, so that multiple DHCP clients and servers may deterministically perform dynamic DNS updates to the same zone.

__init__(*zone*, *fqdn*, **args*, ***kwargs*)

Create an *DHCIDRecord* object

Parameters

- **zone** – Name of zone where the record will be added
- **fqdn** – Name of node where the record will be added
- **digest** – Base-64 encoded digest of DHCP data
- **ttl** – TTL for this record

digest

Base-64 encoded digest of DHCP data

rdata()

Return this *DHCIDRecord*'s rdata as a JSON blob

DNAMERecord

class dyn.tm.records.**DNAMERecord**(*zone*, *fqdn*, **args*, ***kwargs*)

The Delegation of Reverse Name (DNAME) Record is designed to assist the delegation of reverse mapping by reducing the size of the data that must be entered. DNAME's are designed to be used in conjunction with a bit label but do not strictly require one. However, do note that without a bit label a DNAME is equivalent to a CNAME when used in a reverse-map zone file.

__init__(*zone*, *fqdn*, **args*, ***kwargs*)

Create an *DNAMERecord* object

Parameters

- **zone** – Name of zone where the record will be added
- **fqdn** – Name of node where the record will be added
- **dname** – Target Hostname
- **ttl** – TTL for this record

dname
Target hostname

rdata()
Return this *DNAMERecord*'s rdata as a JSON blob

DNSKEYRecord

class dyn.tm.records.**DNSKEYRecord**(*zone*, *fqdn*, **args*, ***kwargs*)
The DNSKEY Record describes the public key of a public key (asymmetric) cryptographic algorithm used with DNSSEC.nis. It is typically used to authenticate signed keys or zones.

__init__(*zone*, *fqdn*, **args*, ***kwargs*)
Create a *DNSKEYRecord* object

Parameters

- **zone** – Name of zone where the record will be added
- **fqdn** – Name of node where the record will be added
- **protocol** – Numeric value for protocol
- **public_key** – The public key for the DNSSEC signed zone
- **algorithm** – Numeric value representing the public key encryption algorithm which will sign the zone. Must be one of 1 (RSA-MD5), 2 (Diffie-Hellman), 3 (DSA/SHA-1), 4 (Elliptic Curve), or 5 (RSA-SHA-1)
- **flags** – Numeric value confirming this is the zone's DNSKEY
- **ttl** – TTL for this record. Use 0 for zone default

algorithm
Public key encryption algorithm will sign the zone

flags
Numeric value confirming this is the zone's DNSKEY

protocol
Numeric value for protocol. Set to 3 for DNSSEC

public_key
The public key for the DNSSEC signed zone

rdata()
Return this *DNSKEYRecord*'s rdata as a JSON blob

DSRecord

class dyn.tm.records.**DSRecord**(*zone*, *fqdn*, **args*, ***kwargs*)

The Delegation Signer (DS) record type is used in DNSSEC to create the chain of trust or authority from a signed parent zone to a signed child zone.

__init__(*zone*, *fqdn*, **args*, ***kwargs*)

Create a *DSRecord* object

Parameters

- **zone** – Name of zone where the record will be added
- **fqdn** – Name of node where the record will be added
- **digest** – The digest in hexadecimal form. 20-byte, hexadecimal-encoded, one-way hash of the DNSKEY record surrounded by parenthesis characters '(' & ')'
- **keytag** – The digest mechanism to use to verify the digest
- **algorithm** – Numeric value representing the public key encryption algorithm which will sign the zone. Must be one of 1 (RSA-MD5), 2 (Diffie-Hellman), 3 (DSA/SHA-1), 4 (Elliptic Curve), or 5 (RSA-SHA-1)
- **digtype** – the digest mechanism to use to verify the digest. Valid values are SHA1, SHA256
- **ttl** – TTL for this record. Use 0 for zone default

algorithm

Identifies the encoding algorithm

digest

The digest in hexadecimal form. 20-byte, hexadecimal-encoded, one-way hash of the DNSKEY record surrounded by parenthesis characters

digtype

Identifies which digest mechanism to use to verify the digest

keytag

Identifies which digest mechanism to use to verify the digest

rdata()

Return this *DSRecord*'s rdata as a JSON blob

KEYRecord

class dyn.tm.records.**KEYRecord**(*zone*, *fqdn*, **args*, ***kwargs*)

"Public Key" (KEY) Records are used for the storage of public keys for use by multiple applications such as IPSec, SSH, etc..., as well as for use by DNS security methods including the original DNSSEC protocol. However, as of RFC3445 the use of *KEYRecord*'s have been limited to use in DNS Security operations such as DDNS and zone transfer due to the difficulty of querying for specific uses.

__init__(*zone*, *fqdn*, **args*, ***kwargs*)

Create a *KEYRecord* object

Parameters

- **zone** – Name of zone where the record will be added
- **fqdn** – Name of node where the record will be added

- **algorithm** – Numeric value representing the public key encryption algorithm which will sign the zone. Must be one of 1 (RSA-MD5), 2 (Diffie-Hellman), 3 (DSA/SHA-1), 4 (Elliptic Curve), or 5 (RSA-SHA-1)
- **flags** – See RFC 2535 for information on KEY record flags
- **protocol** – Numeric identifier of the protocol for this KEY record
- **public_key** – The public key for this record
- **ttl** – TTL for the record in seconds

algorithm
Numeric identifier for algorithm used

flags
See RFC 2535 for information about Key record flags

protocol
Numeric identifier of the protocol for this KEY record

public_key
The public key for this record

rdata()
Return this *KEYRecord*'s rdata as a JSON blob

KXRecord

class dyn.tm.records.**KXRecord**(*zone*, *fqdn*, **args*, ***kwargs*)
The "Key Exchanger" (KX) Record type is provided with one or more alternative hosts.

__init__(*zone*, *fqdn*, **args*, ***kwargs*)
Create a *KXRecord* object

Parameters

- **zone** – Name of zone where the record will be added
- **fqdn** – Name of node where the record will be added
- **exchange** – Hostname that will act as the Key Exchanger. The hostname must have a *CNAMERecord*, an *ARecord* and/or an *AAAARecord* associated with it
- **preference** – Numeric value for priority usage. Lower value takes precedence over higher value where two records of the same type exist on the zone/node
- **ttl** – TTL for the record in seconds

exchange
Hostname that will act as the Key Exchanger. The hostname must have a CNAME record, an A Record and/or an AAAA record associated with it

preference
Numeric value for priority usage. Lower value takes precedence over higher value where two records of the same type exist on the zone/node

rdata()
Return this *KXRecord*'s rdata as a JSON blob

LOCRecord

class dyn.tm.records.**LOCRecord**(*zone*, *fqdn*, **args*, ***kwargs*)

LOCRecord's allow for the definition of geographic positioning information associated with a host or service name.

__init__(*zone*, *fqdn*, **args*, ***kwargs*)

Create a *LOCRecord* object

Parameters

- **zone** – Name of zone where the record will be added
- **fqdn** – Name of node where the record will be added
- **altitude** – Measured in meters above sea level
- **horiz_pre** –
- **latitude** – Measured in degrees, minutes, and seconds with N/S indicator for North and South
- **longitude** – Measured in degrees, minutes, and seconds with E/W indicator for East and West
- **size** –
- **version** –
- **vert_pre** –
- **ttl** – TTL for the record in seconds

altitude

Measured in meters above sea level

horiz_pre

Defaults to 10,000 meters

latitude

Measured in degrees, minutes, and seconds with N/S indicator for North and South. Example: 45 24 15 N, where 45 = degrees, 24 = minutes, 15 = seconds

longitude

Measured in degrees, minutes, and seconds with E/W indicator for East and West. Example 89 23 18 W, where 89 = degrees, 23 = minutes, 18 = seconds

rdata()

Return this *LOCRecord*'s rdata as a JSON blob

size

Defaults to 1 meter

version

Number of the representation. Must be zero (0) NOTE: Version has no setter, because it will never not be 0

vert_pre

IPSECKEYRecord

class dyn.tm.records.**IPSECKEYRecord**(*zone*, *fqdn*, **args*, ***kwargs*)

The IPSECKEY is used for storage of keys used specifically for IPSec oerations

__init__(*zone*, *fqdn*, **args*, ***kwargs*)
Create an *IPSECKEYRecord* object

Parameters

- **zone** – Name of zone where the record will be added
- **fqdn** – Name of node where the record will be added
- **precedence** – Indicates priority among multiple IPSECKEYS. Lower numbers are higher priority
- **gatetype** – Gateway type. Must be one of 0, 1, 2, or 3
- **algorithm** – Public key's cryptographic algorithm and format. Must be one of 0, 1, or 2
- **gateway** – Gateway used to create IPsec tunnel. Based on Gateway type
- **public_key** – Base64 encoding of the public key. Whitespace is allowed
- **ttl** – TTL for the record in seconds

algorithm
Public key's cryptographic algorithm and format

gatetype
Gateway type. Must be one of 0, 1, 2, or 3

gateway
Gateway used to create IPsec tunnel. Based on Gateway type

precedence
Indicates priority among multiple IPSECKEYS. Lower numbers are higher priority

public_key
Base64 encoding of the public key. Whitespace is allowed

rdata()
Return this *IPSECKEYRecord*'s rdata as a JSON blob

MXRecord

class dyn.tm.records.**MXRecord**(*zone*, *fqdn*, **args*, ***kwargs*)
The "Mail Exchanger" record type specifies the name and relative preference of mail servers for a Zone. Defined in RFC 1035

__init__(*zone*, *fqdn*, **args*, ***kwargs*)
Create an *MXRecord* object

Parameters

- **zone** – Name of zone where the record will be added
- **fqdn** – Name of node where the record will be added
- **exchange** – Hostname of the server responsible for accepting mail messages in the zone
- **preference** – Numeric value for priority usage. Lower value takes precedence over higher value where two records of the same type exist on the zone/node.
- **ttl** – TTL for the record in seconds

exchange
Hostname of the server responsible for accepting mail messages in the zone

preference
Numeric value for priority usage. Lower value takes precedence over higher value where two records of the same type exist on the zone/node

rdata()
Return this *MXRecord*'s rdata as a JSON blob

NAPTRRecord

class dyn.tm.records.**NAPTRRecord**(*zone*, *fqdn*, **args*, ***kwargs*)
Naming Authority Pointer Records are a part of the Dynamic Delegation Discovery System (DDDS). The NAPTR is a generic record that defines a *rule* that may be applied to private data owned by a client application.

__init__(*zone*, *fqdn*, **args*, ***kwargs*)
Create an *NAPTRRecord* object

Parameters

- **zone** – Name of zone where the record will be added
- **fqdn** – Name of node where the record will be added
- **order** – Indicates the required priority for processing NAPTR records. Lowest value is used first.
- **preference** – Indicates priority where two or more NAPTR records have identical order values. Lowest value is used first.
- **services** – Always starts with "e2u+" (E.164 to URI). After the e2u+ there is a string that defines the type and optionally the subtype of the URI where this *NAPTRRecord* points.
- **regexp** – The NAPTR record accepts regular expressions
- **replacement** – The next domain name to find. Only applies if this *NAPTRRecord* is non-terminal.
- **flags** – Should be the letter "U". This indicates that this NAPTR record terminal
- **ttl** – TTL for the record in seconds

flags
Should be the letter "U". This indicates that this NAPTR record terminal (E.164 number that maps directly to a URI)

order
Indicates the required priority for processing NAPTR records. Lowest value is used first

preference
Indicates priority where two or more NAPTR records have identical order values. Lowest value is used first.

rdata()
Return this *NAPTRRecord*'s rdata as a JSON blob

regexp
The NAPTR record accepts regular expressions

replacement
The next domain name to find. Only applies if this NAPTR record is non-terminal

services
Always starts with "e2u+" (E.164 to URI). After the e2u+ there is a string that defines the type and optionally the subtype of the URI where this NAPTR record points

PTRRecord

class dyn.tm.records.**PTRRecord**(*zone*, *fqdn*, **args*, ***kwargs*)
Pointer Records are used to reverse map an IPv4 or IPv6 IP address to a host name

__init__(*zone*, *fqdn*, **args*, ***kwargs*)
Create a *PTRRecord* object

Parameters

- **zone** – Name of zone where the record will be added
- **fqdn** – Name of node where the record will be added
- **ptrdname** – The hostname where the IP address should be directed
- **ttl** – TTL for the record in seconds

ptrdname
Hostname where the IP address should be directed

rdata()
Return this *PTRRecord*'s rdata as a JSON blob

PXRecord

class dyn.tm.records.**PXRecord**(*zone*, *fqdn*, **args*, ***kwargs*)
The X.400 to RFC 822 E-mail RR allows mapping of ITU X.400 format e-mail addresses to RFC 822 format e-mail addresses using a MIXER-conformant gateway.

__init__(*zone*, *fqdn*, **args*, ***kwargs*)
Create an *PXRecord* object

Parameters

- **zone** – Name of zone where the record will be added
- **fqdn** – Name of node where the record will be added
- **preference** – Sets priority for processing records of the same type. Lowest value is processed first.
- **map822** – mail hostname
- **mapx400** – The domain name derived from the X.400 part of MCGAM
- **ttl** – TTL for the record in seconds

map822
mail hostname

mapx400
Enter the domain name derived from the X.400 part of MCGAM

preference
Sets priority for processing records of the same type. Lowest value is processed first

rdata()
Return this PXRRecord's rdata as a JSON blob

NSAPRecord

class dyn.tm.records.**NSAPRecord**(*zone*, *fqdn*, **args*, ***kwargs*)
The Network Services Access Point record is the equivalent of an RR for ISO's Open Systems Interconnect (OSI) system in that it maps a host name to an endpoint address.

__init__(*zone*, *fqdn*, **args*, ***kwargs*)
Create an *PXRecord* object

Parameters

- **zone** – Name of zone where the record will be added
- **fqdn** – Name of node where the record will be added
- **nsap** – Hex-encoded NSAP identifier
- **ttl** – TTL for the record in seconds

nsap
Hex-encoded NSAP identifier

rdata()
Return this *NSAPRecord*'s rdata as a JSON blob

RPRecord

class dyn.tm.records.**RPRecord**(*zone*, *fqdn*, **args*, ***kwargs*)
The Respnosible Person record allows an email address and some optional human readable text to be associated with a host. Due to privacy and spam considerations, RPRecords are not widely used on public servers but can provide very useful contact data during diagnosis and debugging network problems.

__init__(*zone*, *fqdn*, **args*, ***kwargs*)
Create an *RPRecord* object

Parameters

- **zone** – Name of zone where the record will be added
- **fqdn** – Name of node where the record will be added
- **mbox** – Email address of the Responsible Person.
- **txtdname** – Hostname where a TXT record exists with more information on the responsible person.
- **ttl** – TTL for the record in seconds

mbox
Email address of the Responsible Person. Data format: Replace @ symbol with a dot '.' in the address

rdata()
Return this *RPRecord*'s rdata as a JSON blob

txtdname
Hostname where a TXT record exists with more information on the responsible person

NSRecord

class dyn.tm.records.**NSRecord**(*zone*, *fqdn*, **args*, ***kwargs*)
Nameserver Records are used to list all the nameservers that will respond authoritatively for a domain.

__init__(*zone*, *fqdn*, **args*, ***kwargs*)
Create an *NSRecord* object

Parameters

- **zone** – Name of zone where the record will be added
- **fqdn** – Name of node where the record will be added
- **nsdname** – Hostname of the authoritative Nameserver for the zone
- **service_class** – Hostname of the authoritative Nameserver for the zone
- **ttl** – TTL for the record in seconds

nsdname
Hostname of the authoritative Nameserver for the zone

rdata()
Return this *NSRecord*'s rdata as a JSON blob

service_class
Hostname of the authoritative Nameserver for the zone

SOARecord

class dyn.tm.records.**SOARecord**(*zone*, *fqdn*, **args*, ***kwargs*)
The Start of Authority Record describes the global properties for the Zone (or domain). Only one SOA Record is allowed under a zone at any given time. NOTE: Dynect users do not have the permissions required to create or delete SOA records on the Dynect System.

__init__(*zone*, *fqdn*, **args*, ***kwargs*)
Create an *SOARecord* object

Parameters

- **zone** – Name of zone where the record will be added
- **fqdn** – Name of node where the record will be added

delete()
Users can not POST or DELETE SOA Records

minimum
The minimum TTL for this *SOARecord*

rdata()
Return this *SOARecord*'s rdata as a JSON blob

rname
Domain name which specifies the mailbox of the person responsible for this zone

serial_style
The style of the zone's serial

ttl
The TTL for this record

SPFRecord

class dyn.tm.records.**SPFRecord**(*zone*, *fqdn*, **args*, ***kwargs*)

Sender Policy Framework Records are used to allow a recieving Message Transfer Agent (MTA) to verify that the originating IP of an email from a sender is authorized to send main for the sender's domain.

__init__(*zone*, *fqdn*, **args*, ***kwargs*)

Create an *SPFRecord* object

Parameters

- **zone** – Name of zone where the record will be added
- **fqdn** – Name of node where the record will be added
- **txtdata** – Free text containing SPF record information
- **ttl** – TTL for the record in seconds

rdata()

Return this *SPFRecord*'s rdata as a JSON blob

txtdata

Free text box containing SPF record information

SRVRecord

class dyn.tm.records.**SRVRecord**(*zone*, *fqdn*, **args*, ***kwargs*)

The Services Record type allow a service to be associated with a host name. A user or application that wishes to discover where a service is located can interrogate for the relevant SRV that describes the service.

__init__(*zone*, *fqdn*, **args*, ***kwargs*)

Create a *SRVRecord* object

Parameters

- **zone** – Name of zone where the record will be added
- **fqdn** – Name of node where the record will be added
- **port** – Indicates the port where the service is running
- **priority** – Numeric value for priority usage. Lower value takes precedence over higher value where two records of the same type exist on the zone/node
- **target** – The domain name of a host where the service is running on the specified port
- **weight** – Secondary prioritizing of records to serve. Records of equal priority should be served based on their weight. Higher values are served more often
- **ttl** – TTL for the record. Set to 0 to use zone default

port

Indicates the port where the service is running

priority

Numeric value for priority usage. Lower value takes precedence over higher value where two records of the same type exist on the zone/node

rdata()

Return this *SRVRecord*'s rdata as a JSON blob

target
The domain name of a host where the service is running on the specified *port*

weight
Secondary prioritizing of records to serve. Records of equal priority should be served based on their weight. Higher values are served more often

TLSARecord

class dyn.tm.records.**TLSARecord**(*zone*, *fqdn*, **args*, ***kwargs*)
The TLSA record is used to associate a TLS server certificate or public key with the domain name where the record is found, thus forming a "TLSA certificate association". Defined in RFC 6698

__init__(*zone*, *fqdn*, **args*, ***kwargs*)
Create an *TLSARecord* object

Parameters

- **zone** – Name of zone where the record will be added
- **fqdn** – Name of node where the record will be added
- **cert_usage** – Specifies the provided association that will be used to match the certificate presented in the TLS handshake. Example values: 0 (CA constraint), 1 (Service certificate constraint), 2 (Trust anchor assertion), 3 (Domain-issued certificate)
- **selector** – Specifies which part of the TLS certificate presented by the server will be matched against the association data. Example values: 0 (Full certificate), 1 (SubjectPublicKeyInfo)
- **match_type** – Specifies how the certificate association is presented. Example values: 0 (No hash used), 1 (SHA-256), 2 (SHA-512)
- **certificate** – Full certificate or its SubjectPublicKeyInfo, or hash based on the matching type.
- **ttl** – TTL for the record in seconds

cert_usage
Specifies the provided association that will be used to match the certificate presented in the TLS handshake

certificate
Full certificate or its SubjectPublicKeyInfo, or hash based on the matching type

match_type
Specifies how the certificate association is presented.

rdata()
Return this *TLSARecord*'s rdata as a JSON blob

selector
Specifies which part of the TLS certificate presented by the server will be matched against the association data.

TXTRecord

class dyn.tm.records.**TXTRecord**(*zone*, *fqdn*, **args*, ***kwargs*)
The Text record type provides the ability to associate arbitrary text with a name. For example, it can be used to provide a description of the host, service contacts, or any other required system information.

__init__(*zone*, *fqdn*, **args*, ***kwargs*)
Create a new TXTRecord object

Parameters

- **zone** – Name of zone where the record will be added
- **fqdn** – Name of node where the record will be added
- **txtdata** – Free form text for this *TXTRecord*
- **ttl** – TTL for the record. Set to 0 to use zone default

rdata()
Return this *TXTRecord*'s rdata as a JSON blob

txtdata
Free form text

Example Record Usage

Below are a few basic examples of how to use some different record types in a variety of ways.

Create a new Record

```
>>> from dyn.tm.records import ARecord
>>> # Create a dyn.tmSession
>>> # Assuming you own the Zone 'example.com'
>>> new_a = ARecord('example.com', 'example.com.', address='127.0.0.1')
```

Getting an Existing Record

Getting records is a slightly more complicated task if you don't have the record id readily accessible. Below is an example which shows the easiest way to get a specific record, assuming you don't have the id readily available.

```
>>> from dyn.tm.zones import Zone
>>> # Create a dyn.tmSession
>>> # Once again, assuming you own 'example.com'
>>> zone = Zone('example.com')
>>> all_records = zone.get_node().get_any_records()
>>> for record in all_records:
...     # Find your record, more info coming soon...
```

Delete all Records

As of v1.4.2 you can also delete all records of a certain type on a specific node

```
>>> from dyn.tm.records import ARecord
>>> my_node = ARecord('myzone.com', 'fqdn.myzone.com.', create=False)
>>> my_node.delete()  # Warning, this will delete ALL ARecords on fqdn.myzone.com.
```

Services

The `services` module contains interfaces to all of the various service management features offered by the dyn.tm REST API

Active Failover

The `services` module contains interfaces to all of the various service management features offered by the dyn.tm REST API

HealthMonitor

class `dyn.tm.services.active_failover.`**`HealthMonitor`**(*protocol*, *interval*, *retries=None*, *timeout=None*, *port=None*, *path=None*, *host=None*, *header=None*, *expected=None*)

A health monitor for an `ActiveFailover` service

`__init__`(*protocol*, *interval*, *retries=None*, *timeout=None*, *port=None*, *path=None*, *host=None*, *header=None*, *expected=None*)

Create a `HealthMonitor` object

Parameters

- **`protocol`** – The protocol to monitor. Must be either HTTP, HTTPS, PING, SMTP, or TCP
- **`interval`** – How often (in minutes) to run this `HealthMonitor`. Must be 1, 5, 10, or 15,
- **`retries`** – The number of retries the monitor attempts on failure before giving up
- **`timeout`** – The amount of time in seconds before the connection attempt times out
- **`port`** – For HTTP(S)/SMTP/TCP probes, an alternate connection port
- **`path`** – For HTTP(S) probes, a specific path to request
- **`host`** – For HTTP(S) probes, a value to pass in to the Host
- **`header`** – For HTTP(S) probes, additional header fields/values to pass in, separated by a newline character.
- **`expected`** – For HTTP(S) probes, a string to search for in the response. For SMTP probes, a string to compare the banner against. Failure to find this string means the monitor will report a down status.

`expected`

For HTTP(S) probes, a string to search for in the response. For SMTP probes, a string to compare the banner against. Failure to find this string means the monitor will report a down status

`header`

For HTTP(S) probes, additional header fields/values to pass in, separated by a newline character

`host`

For HTTP(S) probes, a value to pass in to the Host

`interval`

How often to run this monitor

path
For HTTP(S) probes, a specific path to request

port
For HTTP(S)/SMTP/TCP probes, an alternate connection port

protocol
The protocol to monitor

retries
The number of retries the monitor attempts on failure before giving up

status
Get the current status of this `HealthMonitor` from the DynECT System

timeout
The amount of time in seconds before the connection attempt times out

to_json()
Convert this `HealthMonitor` object to a JSON blob

Active Failover

class dyn.tm.services.active_failover.**ActiveFailover**(*zone*, *fqdn*, **args*, ***kwargs*)
With Active Failover, we monitor your Primary IP. If a failover event is detected, our system auto switches (hot swaps) to your dedicated back-up IP

__init__(*zone*, *fqdn*, **args*, ***kwargs*)
Create a new `ActiveFailover` object

Parameters

- **zone** – The zone to attach this `ActiveFailover` service to
- **fqdn** – The FQDN where this `ActiveFailover` service will be attached
- **address** – IPv4 Address or FQDN being monitored by this `ActiveFailover` service
- **failover_mode** – Indicates the target failover resource type.
- **failover_data** – The IPv4 Address or CNAME data for the failover target
- **auto_recover** – Indicates whether this service should restore its original state when the source IPs resume online status
- **notify_events** – A comma separated list of what events trigger notifications
- **syslog_server** – The Hostname or IP address of a server to receive syslog notifications on monitoring events
- **syslog_port** – The port where the remote syslog server listens
- **syslog_ident** – The ident to use when sending syslog notifications
- **syslog_facility** – The syslog facility to use when sending syslog notifications
- **syslog_delivery** – The syslog delivery action type. 'all' will deliver notifications no matter what the endpoint state. 'change' (default) will deliver only on change in the detected endpoint state
- **monitor** – The `HealthMonitor` for this `ActiveFailover` service

- **contact_nickname** – Name of contact to receive notifications from this ActiveFailover service
- **ttl** – Time To Live in seconds of records in the service. Must be less than 1/2 of the Health Probe's monitoring interval
- **syslog_probe_fmt** – see below for format:
- **syslog_status_fmt** – see below for format: Use the following format for syslog_xxxx_fmt paramaters. %hos hostname %tim current timestamp or monitored interval %reg region code %sta status %ser record serial %rda rdata %sit monitoring site %rti response time %msg message from monitoring %adr address of monitored node %med median value %rts response times (RTTM)
- **recovery_delay** – number of up status polling intervals to consider service up

activate()
Activate this ActiveFailover service

active
Return whether or not this ActiveFailover service is active. When setting directly, rather than using activate/deactivate valid arguments are 'Y' or True to activate, or 'N' or False to deactivate.

Note: If your service is already active and you try to activate it, nothing will happen. And vice versa for deactivation.

Returns An Active object representing the current state of this ActiveFailover Service

address
IPv4 Address or FQDN being monitored by this ActiveFailover service

auto_recover
Indicates whether this service should restore its original state when the source IPs resume online status

contact_nickname
Name of contact to receive notifications from this ActiveFailover service

deactivate()
Deactivate this ActiveFailover service

delete()
Delete this ActiveFailover service from the Dynect System

failover_data
The IPv4 Address or CNAME data for the failover target

failover_mode
Indicates the target failover resource type.

fqdn
The FQDN where this ActiveFailover service will be attached

monitor
The HealthMonitor for this ActiveFailover service

notify_events
A comma separated list of what events trigger notifications

recover()
Recover this ActiveFailover service

recovery_delay

syslog_delivery

syslog_facility
The syslog facility to use when sending syslog notifications

syslog_ident
The ident to use when sending syslog notifications

syslog_port
The port where the remote syslog server listens

syslog_probe_format

syslog_server
The Hostname or IP address of a server to receive syslog notifications on monitoring events

syslog_status_format

task
`Task` for most recent system action on this `ActiveFailover`.

ttl
Time To Live in seconds of records in the service. Must be less than 1/2 of the Health Probe's monitoring interval

zone
The zone to attach this `ActiveFailover` service to

Active Failover Examples

The following examples highlight how to use the `ActiveFailover` class to get/create `ActiveFailover`'s on the dyn.tm System and how to edit these objects from within a Python script.

Creating a new Active Failover Service

The following example shows how to create a new `ActiveFailover` on the dyn.tm System and how to edit some of the fields using the returned `ActiveFailover` object.

```
>>> from dyn.tm.services.active_failover import HealthMonitor, ActiveFailover,
>>> # Create a dyn.tmSession
>>> mon = HealthMonitor(protocol='HTTP', interval='1', expected='Example')
>>> # Assuming you own the zone 'example.com'
>>> afo = ActiveFailover('example.com', 'example.com.', '127.0.0.1', 'ip',
...                      '127.0.0.2', mon, 'mycontact')
>>> afo.notify_events = 'ip, nosrv'
>>> afo.notify_events
u'ip, nosrv'
```

Getting an Existing Active Failover Service

The following example shows how to get an existing `ActiveFailover` from the dyn.tm System and how to edit some of the same fields mentioned above.

```
>>> from dyn.tm.services.active_failover import HealthMonitor, ActiveFailover,
>>> # Create a dyn.tmSession
>>> # Once again, assuming you own 'example.com'
```

```
>>> afo = ActiveFailover('example.com', 'example.com.')
>>> afo.active
u'Y'
>>> afo.deactivate()
>>> afo.active
u'N'
```

Dynamic DNS

class dyn.tm.services.ddns.**DynamicDNS**(*zone*, *fqdn*, **args*, ***kwargs*)

DynamicDNS is a service which aliases a dynamic IP Address to a static hostname

__init__(*zone*, *fqdn*, **args*, ***kwargs*)

Create a new DynamicDNS service object

Parameters

- **zone** – The zone to attach this DDNS Service to
- **fqdn** – The FQDN of the node where this service will be attached
- **record_type** – Either A, for IPv4, or AAAA, for IPv6
- **address** – IPv4 or IPv6 address for the service
- **full_setup** – Flag to indicate a user is specified
- **user** – Name of the user to create, or the name of an existing update user to allow access to this service

activate()

Activate this Dynamic DNS service

active

Returns whether or not this DynamicDNS Service is currently active. When setting directly, rather than using activate/deactivate valid arguments are 'Y' or True to activate, or 'N' or False to deactivate. Note: If your service is already active and you try to activate it, nothing will happen. And vice versa for deactivation.

Returns An Active object representing the current state of this DynamicDNS Service

address

IPv4 or IPv6 address for this DynamicDNS service

deactivate()

Deactivate this Dynamic DNS service

delete()

Delete this Dynamic DNS service from the DynECT System

fqdn

The fqdn that this DynamicDNS Service is attached to is a read-only attribute

record_type

The record_type of a DDNS Service is a read-only attribute

reset()

Resets the abuse count on this Dynamic DNS service

user

The User attribute of a DDNS Service is a read-only attribute

zone
The zone that this DynamicDNS Service is attached to is a read-only attribute

Dynamic DNS Examples

The following examples highlight how to use the `DynamicDNS` class to get/create `DynamicDNS`'s on the dyn.tm System and how to edit these objects from within a Python script.

Creating a new Dynamic DNS Service

The following example shows how to create a new `DynamicDNS` on the dyn.tm System and how to edit some of the fields using the returned `DynamicDNS` object.

```
>>> from dyn.tm.services.ddns import DynamicDNS
>>> # Create a dyn.tmSession
>>> # Assuming you own the zone 'example.com'
>>> dyndns = DynamicDNS('example.com', 'example.com.', 'A', '127.0.0.1')
>>> dyndns.ttl = 180
>>> dyndns.ttl
180
>>> dyndns.record_type
u'A'
```

Getting an Existing Dynamic DNS Service

The following example shows how to get an existing `DynamicDNS` from the dyn.tm System and how to edit some of the same fields mentioned above.

```
>>> from dyn.tm.services.ddns import DynamicDNS
>>> # Create a dyn.tmSession
>>> # Once again, assuming you own 'example.com'
>>> dyndns = DynamicDNS('example.com', 'example.com.', record_type='A')
>>> dyndns.active
u'Y'
>>> dyndns.deactivate()
>>> dyndns.active
u'N'
```

DNSSEC

The `services` module contains interfaces to all of the various service management features offered by the dyn.tm REST API

List Functions

The following function is primarily a helper function which performs an API "Get All" call. This function returns a single `list` of `DNSSEC` service objects.

`dyn.tm.services.dnssec.`**`get_all_dnssec`**`()`

Returns A `list` of `DNSSEC` Services

Classes

DNSSECKey

class dyn.tm.services.dnssec.**DNSSECKey**(*key_type*, *algorithm*, *bits*, *start_ts=None*, *lifetime=None*, *overlap=None*, *expire_ts=None*, ***kwargs*)

A Key used by the DNSSEC service

__init__(*key_type*, *algorithm*, *bits*, *start_ts=None*, *lifetime=None*, *overlap=None*, *expire_ts=None*, ***kwargs*)

Create a DNSSECKey object

Parameters

- **key_type** – The type of this key. (KSK or ZSK)
- **algorithm** – One of (RSA/SHA-1, RSA/SHA-256, RSA/SHA-512, DSA, ECDSAP256SHA256, ECDSAP384SHA384)
- **bits** – length of the key. Valid values: 256, 384, 1024, 2048, or 4096
- **start_ts** – An epoch time when key is to be valid
- **lifetime** – Lifetime of the key expressed in seconds
- **overlap** – Time before key expiration when a replacement key is prepared, expressed in seconds. Default = 7 days.
- **expire_ts** – An epoch time when this key is to expire
- **dnskey** – The KSK or ZSK record data
- **ds** – One of the DS records for the KSK. ZSKs will have this value intialized, but with null values.
- **all_ds** – All the DS records associated with this KSK. Applies only to KSK, ZSK will have a zero-length list.

DNSSEC

class dyn.tm.services.dnssec.**DNSSEC**(*zone*, **args*, ***kwargs*)

A DynECT System DNSSEC Service

__init__(*zone*, **args*, ***kwargs*)

Create a DNSSEC object

Parameters

- **zone** – the zone this service will be attached to
- **keys** – a list of DNSSECKey's for the service
- **contact_nickname** – Name of contact to receive notifications
- **notify_events** – A list of events that trigger notifications. Valid values are "create" (a new version of a key was created), "expire" (a key was automatically expired), or "warning" (early warnings (2 weeks, 1 week, 1 day) of events)

activate()

Activate this DNSSEC service

active
The current status of this DNSSEC service. When setting directly, rather than using activate/deactivate valid arguments are 'Y' or True to activate, or 'N' or False to deactivate. Note: If your service is already active and you try to activate it, nothing will happen. And vice versa for deactivation.

Returns An Active object representing the current state of this DNSSEC Service

contact_nickname
Name of contact to receive notifications

deactivate()
Deactivate this DNSSEC service

delete()
Delete this DNSSEC Service from the DynECT System

keys
A List of DNSSECKey's associated with this DNSSEC service

notify_events
A list of events that trigger notifications. Valid values are: create (a new version of a key was created), expire (a key was automatically expired), warning (early warnings (2 weeks, 1 week, 1 day) of events)

timeline_report(*start_ts=None*, *end_ts=None*)
Generates a report of events this DNSSEC service has performed and has scheduled to perform

Parameters

- **start_ts** – datetime.datetime instance identifying point in time for the start of the timeline report
- **end_ts** – datetime.datetime instance identifying point in time for the end of the timeline report. Defaults to datetime.datetime.now()

zone
The name of the zone where this service exists. This is a read-only property

DNSSEC Examples

The following examples highlight how to use the DNSSEC class to get/create DNSSEC's on the dyn.tm System and how to edit these objects from within a Python script.

Creating a new DNSSEC Service

The following example shows how to create a new DNSSEC on the dyn.tm System and how to edit some of the fields using the returned DNSSEC object.

```
>>> from dyn.tm.services.dnssec import DNSSECKey, DNSSEC
>>> # Create a dyn.tmSession
>>> key1 = DNSSECKey('KSK', 'RSA/SHA-1', 1024)
>>> key2 = DNSSECKey('ZSK', 'RSA/SHA-1', 2048)
>>> # Assuming you own the zone 'example.com'
>>> dnssec = DNSSEC('example.com', [key1, key2], 'mycontactnickname')
>>> dnssec.deactivate()
>>> dnssec.active
u'N'
```

Getting an Existing DNSSEC Service

The following example shows how to get an existing `DNSSEC` from the dyn.tm System and how to edit some of the same fields mentioned above.

```
>>> from dyn.tm.services.dnssec import DNSSEC
>>> # Create a dyn.tmSession
>>> # Once again, assuming you own 'example.com'
>>> dnssec = DNSSEC('example.com', [key1, key2], 'mycontactnickname')
>>> if dnssec.active == 'N':
...     dnssec.activate()
>>> from pprint import pprint
>>> pprint(dnssec.timeline_report())
{}
```

Managing Your DNSSEC Keys

The following example shows how to manage an existing `DNSSEC` services `DNSSECKey`'s.

```
>>> from dyn.tm.services.dnssec import DNSSEC
>>> dnssec = DNSSEC('example.com')
>>> dnssec.keys
[<__main__.DNSSECKey object at 0x10ca84550>, <__main__.DNSSECKey object at␣
↪0x10ca84590>]
>>> new_key = DNSSECKey('ZSK', 'RSA/SHA-1', 1024)
>>> # You must always have two keys, so we add a new one first
>>> dnssec.keys.append(new_key)
>>> # Now that we have two keys we can delete an onld KSK we don't want
>>> for index, key in enumerate(dnssec.keys):
...     if key.key_type == 'KSK' and key.bits == 1024:
...         del dnssec.keys[index]
...         break
>>> dnssec.keys
[<__main__.DNSSECKey object at 0x10ca84590>, <__main__.DNSSECKey object at␣
↪0x10ca78b50>]
```

Traffic Director

The `services` module contains interfaces to all of the various service management features offered by the dyn.tm REST API

List Functions

The following function is primarily a helper function which performs an API "Get All" call. This function returns a single `list` of `TrafficDirector` service objects.

`dyn.tm.services.dsf.`**`get_all_dsf_services`**`()`

Returns A `list` of `TrafficDirector` Services

`dyn.tm.services.dsf.`**`get_all_dsf_monitors`**`()`

Returns A `list` of `DSFMonitor` Services

`dyn.tm.services.dsf.`**`get_all_notifiers`**`()`

Returns A `list` of `DSFNotifier` Services

`dyn.tm.services.dsf.`**`get_all_records`**(*service*)

Parameters **`service`** – a dsf_id string, or `TrafficDirector`

Returns A `list` of `DSFRecord`s from the passed in `service`

Warning! This query may take a long time to run with services with many records!

`dyn.tm.services.dsf.`**`get_all_record_sets`**(*service*)

Parameters **`service`** – a dsf_id string, or `TrafficDirector`

Returns A `list` of `DSFRecordSets` from the passed in *service*

`dyn.tm.services.dsf.`**`get_all_failover_chains`**(*service*)

Parameters **`service`** – a dsf_id string, or `TrafficDirector`

Returns A `list` of `DSFFailoverChains` from the passed in *service*

`dyn.tm.services.dsf.`**`get_all_response_pools`**(*service*)

Parameters **`service`** – a dsf_id string, or `TrafficDirector`

Returns A `list` of `DSFResponsePools` from the passed in *service*

`dyn.tm.services.dsf.`**`get_all_rulesets`**(*service*)

Parameters **`service`** – a dsf_id string, or `TrafficDirector`

Returns A `list` of `DSFRulesets` from the passed in *service*

Classes

DSFRecords

class `dyn.tm.services.dsf.`**`DSFARecord`**(*address*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

An `ARecord` object which is able to store additional data for use by a `TrafficDirector` service.

`__init__`(*address*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

Create a `DSFARecord` object

Parameters

- **`address`** – IPv4 address for the record
- **`ttl`** – TTL for this record
- **`label`** – A unique label for this `DSFARecord`
- **`weight`** – Weight for this `DSFARecord`
- **`automation`** – Defines how eligible can be changed in response to monitoring. Must be one of 'auto', 'auto_down', or 'manual'
- **`endpoints`** – Endpoints are used to determine status, torpidity, and eligible in response to monitor data

- **endpoint_up_count** – Number of endpoints that must be up for the Record status to be 'up'
- **eligible** – Indicates whether or not the Record can be served

class dyn.tm.services.dsf.**DSFAAAARecord**(*address*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

An AAAARecord object which is able to store additional data for use by a TrafficDirector service.

__init__(*address*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

Create a DSFAAAARecord object

Parameters

- **address** – IPv6 address for the record
- **ttl** – TTL for this record
- **label** – A unique label for this DSFAAAARecord
- **weight** – Weight for this DSFAAAARecord
- **automation** – Defines how eligible can be changed in response to monitoring. Must be one of 'auto', 'auto_down', or 'manual'
- **endpoints** – Endpoints are used to determine status, torpidity, and eligible in response to monitor data
- **endpoint_up_count** – Number of endpoints that must be up for the Record status to be 'up'
- **eligible** – Indicates whether or not the Record can be served

class dyn.tm.services.dsf.**DSFALIASRecord**(*alias*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

An AliasRecord object which is able to store additional data for use by a TrafficDirector service.

__init__(*alias*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

Create a DSFALIASRecord object

Parameters

- **alias** – alias target name
- **ttl** – TTL for this record
- **label** – A unique label for this DSFALIASRecord
- **weight** – Weight for this DSFALIASRecord
- **automation** – Defines how eligible can be changed in response to monitoring. Must be one of 'auto', 'auto_down', or 'manual'
- **endpoints** – Endpoints are used to determine status, torpidity, and eligible in response to monitor data
- **endpoint_up_count** – Number of endpoints that must be up for the Record status to be 'up'
- **eligible** – Indicates whether or not the Record can be served

class dyn.tm.services.dsf.**DSFCERTRecord**(*format*, *tag*, *algorithm*, *certificate*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

An CERTRecord object which is able to store additional data for use by a TrafficDirector service.

__init__(*format*, *tag*, *algorithm*, *certificate*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

Create a DSFCERTRecord object

Parameters

- **format** – Numeric value for the certificate type
- **tag** – Numeric value for the public key certificate
- **algorithm** – Public key algorithm number used to generate the certificate
- **certificate** – The public key certificate
- **ttl** – TTL for this record
- **label** – A unique label for this DSFCERTRecord
- **weight** – Weight for this DSFCERTRecord
- **automation** – Defines how eligible can be changed in response to monitoring. Must be one of 'auto', 'auto_down', or 'manual'
- **endpoints** – Endpoints are used to determine status, torpidity, and eligible in response to monitor data
- **endpoint_up_count** – Number of endpoints that must be up for the Record status to be 'up'
- **eligible** – Indicates whether or not the Record can be served

class dyn.tm.services.dsf.**DSFCNAMERecord**(*cname*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

An CNAMERecord object which is able to store additional data for use by a TrafficDirector service.

__init__(*cname*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

Create a DSFCNAMERecord object

Parameters

- **cname** – Hostname
- **ttl** – TTL for this record
- **label** – A unique label for this DSFCNAMERecord
- **weight** – Weight for this DSFCNAMERecord
- **automation** – Defines how eligible can be changed in response to monitoring. Must be one of 'auto', 'auto_down', or 'manual'
- **endpoints** – Endpoints are used to determine status, torpidity, and eligible in response to monitor data
- **endpoint_up_count** – Number of endpoints that must be up for the Record status to be 'up'
- **eligible** – Indicates whether or not the Record can be served

class dyn.tm.services.dsf.**DSFDHCIDRecord**(*digest*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

An DHCIDRecord object which is able to store additional data for use by a TrafficDirector service.

__init__(*digest*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

Create a DSFDHCIDRecord object

Parameters

- **digest** – Base-64 encoded digest of DHCP data
- **ttl** – TTL for this record
- **label** – A unique label for this DSFDHCIDRecord
- **weight** – Weight for this DSFDHCIDRecord
- **automation** – Defines how eligible can be changed in response to monitoring. Must be one of 'auto', 'auto_down', or 'manual'
- **endpoints** – Endpoints are used to determine status, torpidity, and eligible in response to monitor data
- **endpoint_up_count** – Number of endpoints that must be up for the Record status to be 'up'
- **eligible** – Indicates whether or not the Record can be served

class dyn.tm.services.dsf.**DSFDNAMERecord**(*dname*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

An DNAMERecord object which is able to store additional data for use by a TrafficDirector service.

__init__(*dname*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

Create a DSFDNAMERecord object

Parameters

- **dname** – Target Hostname
- **ttl** – TTL for this record
- **label** – A unique label for this DSFDNAMERecord
- **weight** – Weight for this DSFDNAMERecord
- **automation** – Defines how eligible can be changed in response to monitoring. Must be one of 'auto', 'auto_down', or 'manual'
- **endpoints** – Endpoints are used to determine status, torpidity, and eligible in response to monitor data
- **endpoint_up_count** – Number of endpoints that must be up for the Record status to be 'up'
- **eligible** – Indicates whether or not the Record can be served

class dyn.tm.services.dsf.**DSFDNSKEYRecord**(*protocol*, *public_key*, *algorithm=5*, *flags=256*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

An DNSKEYRecord object which is able to store additional data for use by a TrafficDirector service.

__init__(*protocol*, *public_key*, *algorithm=5*, *flags=256*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)
Create a `DSFDNSKEYRecord` object

Parameters

- **protocol** – Numeric value for protocol
- **public_key** – The public key for the DNSSEC signed zone
- **algorithm** – Numeric value representing the public key encryption algorithm which will sign the zone. Must be one of 1 (RSA-MD5), 2 (Diffie-Hellman), 3 (DSA/SHA-1), 4 (Elliptic Curve), or 5 (RSA-SHA-1)
- **flags** – Numeric value confirming this is the zone's DNSKEY
- **ttl** – TTL for this record
- **label** – A unique label for this `DSFDNSKEYRecord`
- **weight** – Weight for this `DSFDNSKEYRecord`
- **automation** – Defines how eligible can be changed in response to monitoring. Must be one of 'auto', 'auto_down', or 'manual'
- **endpoints** – Endpoints are used to determine status, torpidity, and eligible in response to monitor data
- **endpoint_up_count** – Number of endpoints that must be up for the Record status to be 'up'
- **eligible** – Indicates whether or not the Record can be served

class `dyn.tm.services.dsf.`**DSFDSRecord**(*digest*, *keytag*, *algorithm=5*, *digtype=1*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)
An `DSRecord` object which is able to store additional data for use by a `TrafficDirector` service.

__init__(*digest*, *keytag*, *algorithm=5*, *digtype=1*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)
Create a `DSFDSRecord` object

Parameters

- **digest** – The digest in hexadecimal form. 20-byte, hexadecimal-encoded, one-way hash of the DNSKEY record surrounded by parenthesis characters '(' & ')'
- **keytag** – The digest mechanism to use to verify the digest
- **algorithm** – Numeric value representing the public key encryption algorithm which will sign the zone. Must be one of 1 (RSA-MD5), 2 (Diffie-Hellman), 3 (DSA/SHA-1), 4 (Elliptic Curve), or 5 (RSA-SHA-1)
- **digtype** – the digest mechanism to use to verify the digest. Valid values are SHA1, SHA256
- **ttl** – TTL for this record
- **label** – A unique label for this `DSFDSRecord`
- **weight** – Weight for this `DSFDSRecord`
- **automation** – Defines how eligible can be changed in response to monitoring. Must be one of 'auto', 'auto_down', or 'manual'

- **endpoints** – Endpoints are used to determine status, torpidity, and eligible in response to monitor data
- **endpoint_up_count** – Number of endpoints that must be up for the Record status to be 'up'
- **eligible** – Indicates whether or not the Record can be served

class dyn.tm.services.dsf.**DSFKEYRecord**(*algorithm*, *flags*, *protocol*, *public_key*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

An KEYRecord object which is able to store additional data for use by a TrafficDirector service.

__init__(*algorithm*, *flags*, *protocol*, *public_key*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

Create a DSFKEYRecord object

Parameters

- **algorithm** – Numeric value representing the public key encryption algorithm which will sign the zone. Must be one of 1 (RSA-MD5), 2 (Diffie-Hellman), 3 (DSA/SHA-1), 4 (Elliptic Curve), or 5 (RSA-SHA-1)
- **flags** – See RFC 2535 for information on KEY record flags
- **protocol** – Numeric identifier of the protocol for this KEY record
- **public_key** – The public key for this record
- **ttl** – TTL for this record
- **label** – A unique label for this DSFKEYRecord
- **weight** – Weight for this DSFKEYRecord
- **automation** – Defines how eligible can be changed in response to monitoring. Must be one of 'auto', 'auto_down', or 'manual'
- **endpoints** – Endpoints are used to determine status, torpidity, and eligible in response to monitor data
- **endpoint_up_count** – Number of endpoints that must be up for the Record status to be 'up'
- **eligible** – Indicates whether or not the Record can be served

class dyn.tm.services.dsf.**DSFKXRecord**(*exchange*, *preference*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

An KXRecord object which is able to store additional data for use by a TrafficDirector service.

__init__(*exchange*, *preference*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

Create a DSFKXRecord object

Parameters

- **exchange** – Hostname that will act as the Key Exchanger. The hostname must have a CNAMERecord, an ARecord and/or an AAAARecord associated with it
- **preference** – Numeric value for priority usage. Lower value takes precedence over higher value where two records of the same type exist on the zone/node
- **ttl** – TTL for this record

- **label** – A unique label for this DSFKXRecord
- **weight** – Weight for this DSFKXRecord
- **automation** – Defines how eligible can be changed in response to monitoring. Must be one of 'auto', 'auto_down', or 'manual'
- **endpoints** – Endpoints are used to determine status, torpidity, and eligible in response to monitor data
- **endpoint_up_count** – Number of endpoints that must be up for the Record status to be 'up'
- **eligible** – Indicates whether or not the Record can be served

class dyn.tm.services.dsf.**DSFLOCRecord**(*altitude*, *latitude*, *longitude*, *horiz_pre=10000*, *size=1*, *vert_pre=10*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

An LOCRecord object which is able to store additional data for use by a TrafficDirector service.

__init__(*altitude*, *latitude*, *longitude*, *horiz_pre=10000*, *size=1*, *vert_pre=10*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

Create a DSFLOCRecord object

Parameters

- **altitude** – Measured in meters above sea level
- **horiz_pre** –
- **latitude** – Measured in degrees, minutes, and seconds with N/S indicator for North and South
- **longitude** – Measured in degrees, minutes, and seconds with E/W indicator for East and West
- **size** –
- **version** –
- **vert_pre** –
- **ttl** – TTL for this record
- **label** – A unique label for this DSFLOCRecord
- **weight** – Weight for this DSFLOCRecord
- **automation** – Defines how eligible can be changed in response to monitoring. Must be one of 'auto', 'auto_down', or 'manual'
- **endpoints** – Endpoints are used to determine status, torpidity, and eligible in response to monitor data
- **endpoint_up_count** – Number of endpoints that must be up for the Record status to be 'up'
- **eligible** – Indicates whether or not the Record can be served

class dyn.tm.services.dsf.**DSFIPSECKEYRecord**(*precedence*, *gatetype*, *algorithm*, *gateway*, *public_key*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

An `IPSECKEYRecord` object which is able to store additional data for use by a `TrafficDirector` service.

__init__(*precedence*, *gatetype*, *algorithm*, *gateway*, *public_key*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)
Create a `DSFIPSECKEYRecord` object

Parameters

- **precedence** – Indicates priority among multiple IPSECKEYS. Lower numbers are higher priority
- **gatetype** – Gateway type. Must be one of 0, 1, 2, or 3
- **algorithm** – Public key's cryptographic algorithm and format. Must be one of 0, 1, or 2
- **gateway** – Gateway used to create IPsec tunnel. Based on Gateway type
- **public_key** – Base64 encoding of the public key. Whitespace is allowed
- **ttl** – TTL for this record
- **label** – A unique label for this `DSFIPSECKEYRecord`
- **weight** – Weight for this `DSFIPSECKEYRecord`
- **automation** – Defines how eligible can be changed in response to monitoring. Must be one of 'auto', 'auto_down', or 'manual'
- **endpoints** – Endpoints are used to determine status, torpidity, and eligible in response to monitor data
- **endpoint_up_count** – Number of endpoints that must be up for the Record status to be 'up'
- **eligible** – Indicates whether or not the Record can be served

class `dyn.tm.services.dsf.`**DSFMXRecord**(*exchange*, *preference=10*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)
An `MXRecord` object which is able to store additional data for use by a `TrafficDirector` service.

__init__(*exchange*, *preference=10*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)
Create a `DSFMXRecord` object

Parameters

- **exchange** – Hostname of the server responsible for accepting mail messages in the zone
- **preference** – Numeric value for priority usage. Lower value takes precedence over higher value where two records of the same type exist on the zone/node.
- **ttl** – TTL for this record
- **label** – A unique label for this `DSFMXRecord`
- **weight** – Weight for this `DSFMXRecord`
- **automation** – Defines how eligible can be changed in response to monitoring. Must be one of 'auto', 'auto_down', or 'manual'
- **endpoints** – Endpoints are used to determine status, torpidity, and eligible in response to monitor data

- **endpoint_up_count** – Number of endpoints that must be up for the Record status to be 'up'
- **eligible** – Indicates whether or not the Record can be served

class dyn.tm.services.dsf.**DSFNAPTRRecord**(*order*, *preference*, *services*, *regexp*, *replacement*, *flags='U'*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

An NAPTRRecord object which is able to store additional data for use by a TrafficDirector service.

__init__(*order*, *preference*, *services*, *regexp*, *replacement*, *flags='U'*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

Create a DSFNAPTRRecord object

Parameters

- **order** – Indicates the required priority for processing NAPTR records. Lowest value is used first.
- **preference** – Indicates priority where two or more NAPTR records have identical order values. Lowest value is used first.
- **services** – Always starts with "e2u+" (E.164 to URI). After the e2u+ there is a string that defines the type and optionally the subtype of the URI where this NAPTRRecord points.
- **regexp** – The NAPTR record accepts regular expressions
- **replacement** – The next domain name to find. Only applies if this NAPTRRecord is non-terminal.
- **flags** – Should be the letter "U". This indicates that this NAPTR record terminal
- **ttl** – TTL for this record
- **label** – A unique label for this DSFNAPTRRecord
- **weight** – Weight for this DSFNAPTRRecord
- **automation** – Defines how eligible can be changed in response to monitoring. Must be one of 'auto', 'auto_down', or 'manual'
- **endpoints** – Endpoints are used to determine status, torpidity, and eligible in response to monitor data
- **endpoint_up_count** – Number of endpoints that must be up for the Record status to be 'up'
- **eligible** – Indicates whether or not the Record can be served

class dyn.tm.services.dsf.**DSFPTRRecord**(*ptrdname*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

An PTRRecord object which is able to store additional data for use by a TrafficDirector service.

__init__(*ptrdname*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

Create a DSFPTRRecord object

Parameters

- **ptrdname** – The hostname where the IP address should be directed
- **ttl** – TTL for this record

- **label** – A unique label for this DSFPTRRecord
- **weight** – Weight for this DSFPTRRecord
- **automation** – Defines how eligible can be changed in response to monitoring. Must be one of 'auto', 'auto_down', or 'manual'
- **endpoints** – Endpoints are used to determine status, torpidity, and eligible in response to monitor data
- **endpoint_up_count** – Number of endpoints that must be up for the Record status to be 'up'
- **eligible** – Indicates whether or not the Record can be served

class dyn.tm.services.dsf.**DSFPXRecord**(*preference*, *map822*, *mapx400*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

An PXRecord object which is able to store additional data for use by a TrafficDirector service.

__init__(*preference*, *map822*, *mapx400*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

Create a DSFPXRecord object

Parameters

- **preference** – Sets priority for processing records of the same type. Lowest value is processed first.
- **map822** – mail hostname
- **mapx400** – The domain name derived from the X.400 part of MCGAM
- **ttl** – TTL for this record
- **label** – A unique label for this DSFPXRecord
- **weight** – Weight for this DSFPXRecord
- **automation** – Defines how eligible can be changed in response to monitoring. Must be one of 'auto', 'auto_down', or 'manual'
- **endpoints** – Endpoints are used to determine status, torpidity, and eligible in response to monitor data
- **endpoint_up_count** – Number of endpoints that must be up for the Record status to be 'up'
- **eligible** – Indicates whether or not the Record can be served

class dyn.tm.services.dsf.**DSFNSAPRecord**(*nsap*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

An NSAPRecord object which is able to store additional data for use by a TrafficDirector service.

__init__(*nsap*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

Create a DSFNSAPRecord object

Parameters

- **nsap** – Hex-encoded NSAP identifier
- **ttl** – TTL for this record
- **label** – A unique label for this DSFNSAPRecord

- **weight** – Weight for this DSFNSAPRecord
- **automation** – Defines how eligible can be changed in response to monitoring. Must be one of 'auto', 'auto_down', or 'manual'
- **endpoints** – Endpoints are used to determine status, torpidity, and eligible in response to monitor data
- **endpoint_up_count** – Number of endpoints that must be up for the Record status to be 'up'
- **eligible** – Indicates whether or not the Record can be served

class dyn.tm.services.dsf.**DSFRPRecord**(*mbox*, *txtdname*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

An RPRecord object which is able to store additional data for use by a TrafficDirector service.

__init__(*mbox*, *txtdname*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

Create a DSFRPRecord object

Parameters

- **mbox** – Email address of the Responsible Person.
- **txtdname** – Hostname where a TXT record exists with more information on the responsible person.
- **ttl** – TTL for this record
- **label** – A unique label for this DSFRPRecord
- **weight** – Weight for this DSFRPRecord
- **automation** – Defines how eligible can be changed in response to monitoring. Must be one of 'auto', 'auto_down', or 'manual'
- **endpoints** – Endpoints are used to determine status, torpidity, and eligible in response to monitor data
- **endpoint_up_count** – Number of endpoints that must be up for the Record status to be 'up'
- **eligible** – Indicates whether or not the Record can be served

class dyn.tm.services.dsf.**DSFNSRecord**(*nsdname*, *service_class=''*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

An NSRecord object which is able to store additional data for use by a TrafficDirector service.

__init__(*nsdname*, *service_class=''*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

Create a DSFNSRecord object

Parameters

- **nsdname** – Hostname of the authoritative Nameserver for the zone
- **service_class** – Hostname of the authoritative Nameserver for the zone
- **ttl** – TTL for this record
- **label** – A unique label for this DSFNSRecord
- **weight** – Weight for this DSFNSRecord

- **automation** – Defines how eligible can be changed in response to monitoring. Must be one of 'auto', 'auto_down', or 'manual'
- **endpoints** – Endpoints are used to determine status, torpidity, and eligible in response to monitor data
- **endpoint_up_count** – Number of endpoints that must be up for the Record status to be 'up'
- **eligible** – Indicates whether or not the Record can be served

class dyn.tm.services.dsf.**DSFSPFRecord**(*txtdata*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

An SPFRecord object which is able to store additional data for use by a TrafficDirector service.

__init__(*txtdata*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

Create a DSFSPFRecord object

Parameters

- **txtdata** – Free text containing SPF record information
- **ttl** – TTL for this record
- **label** – A unique label for this DSFSPFRecord
- **weight** – Weight for this DSFSPFRecord
- **automation** – Defines how eligible can be changed in response to monitoring. Must be one of 'auto', 'auto_down', or 'manual'
- **endpoints** – Endpoints are used to determine status, torpidity, and eligible in response to monitor data
- **endpoint_up_count** – Number of endpoints that must be up for the Record status to be 'up'
- **eligible** – Indicates whether or not the Record can be served

class dyn.tm.services.dsf.**DSFSRVRecord**(*port*, *priority*, *target*, *rr_weight*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

An SRVRecord object which is able to store additional data for use by a TrafficDirector service.

__init__(*port*, *priority*, *target*, *rr_weight*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

Create a DSFSRVRecord object

Parameters

- **port** – Indicates the port where the service is running
- **priority** – Numeric value for priority usage. Lower value takes precedence over higher value where two records of the same type exist on the zone/node
- **target** – The domain name of a host where the service is running on the specified port
- **rr_weight** – Secondary prioritizing of records to serve. Records of equal priority should be served based on their weight. Higher values are served more often
- **ttl** – TTL for this record
- **label** – A unique label for this DSFSRVRecord

- **weight** – Weight for this DSFSRVRecord
- **automation** – Defines how eligible can be changed in response to monitoring. Must be one of 'auto', 'auto_down', or 'manual'
- **endpoints** – Endpoints are used to determine status, torpidity, and eligible in response to monitor data
- **endpoint_up_count** – Number of endpoints that must be up for the Record status to be 'up'
- **eligible** – Indicates whether or not the Record can be served

class dyn.tm.services.dsf.**DSFSSHFPRecord**(*fptype*, *algorithm*, *fingerprint*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

An SSHFPRecord object which is able to store additional data for use by a TrafficDirector service.

__init__(*fptype*, *algorithm*, *fingerprint*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

Create a DSFSSHFPRecord object

Parameters

- **algorithm** – Numeric value representing the public key encryption algorithm which will sign the zone.
- **fptype** – FingerPrint Type
- **fingerprint** – fingerprint value
- **ttl** – TTL for this record
- **label** – A unique label for this DSFSSHFPRecord
- **weight** – Weight for this DSFSSHFPRecord
- **automation** – Defines how eligible can be changed in response to monitoring. Must be one of 'auto', 'auto_down', or 'manual'
- **endpoints** – Endpoints are used to determine status, torpidity, and eligible in response to monitor data
- **endpoint_up_count** – Number of endpoints that must be up for the Record status to be 'up'
- **eligible** – Indicates whether or not the Record can be served

class dyn.tm.services.dsf.**DSFTXTRecord**(*txtdata*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

An TXTRecord object which is able to store additional data for use by a TrafficDirector service.

__init__(*txtdata*, *ttl=0*, *label=None*, *weight=1*, *automation='auto'*, *endpoints=None*, *endpoint_up_count=None*, *eligible=True*, ***kwargs*)

Create a DSFTXTRecord object

Parameters

- **txtdata** – Plain text data to be served by this DSFTXTRecord
- **ttl** – TTL for this record
- **label** – A unique label for this DSFTXTRecord
- **weight** – Weight for this DSFTXTRecord

- **automation** – Defines how eligible can be changed in response to monitoring. Must be one of 'auto', 'auto_down', or 'manual'
- **endpoints** – Endpoints are used to determine status, torpidity, and eligible in response to monitor data
- **endpoint_up_count** – Number of endpoints that must be up for the Record status to be 'up'
- **eligible** – Indicates whether or not the Record can be served

DSFRecord Examples

The following examples highlight how to use the `DSFRecord` classes to get/create/update/delete `DSFRecord`'s on the dyn.tm System and how to edit these objects from within a Python script. We'll stick to a simple `DSFARecord` in our examples.

Create DSF__Record

We'll assume you already have a `DSFRecordset` object called `record_set` in existence for this example.

```
>>> from dyn.tm.services.dsf import DSFARecord
>>> record = DSFARecord('10.1.1.1', label='TEST RECORD', weight=1, automation='auto',␣
↪eligible=True)
>>> #Now, we create this A record by adding it to an existing record_set
>>> record.add_to_record_set(record_set) #This is automatically published.
```

Update DSF__Record

To change the record IP address of the record we just created, we can use one of our setters.

```
>>> record.address = '20.1.1.1' #This gets published implicitly
>>> #Check to see if it really changed.
>>> record.address
>>>'20.1.1.1'
```

Implicit publishing can be turned off for any object if that is undesirable, check `Modifying Traffic Director Service Properties` below for an example and explanation

Get All DSF__Record

To get all `DSFRecord`: from a certain `TrafficDirector`:

```
>>> from dyn.tm.services.dsf import get_all_records
>>> #Pass in a :class:`TrafficDirector`: instance to the following call:
>>> get_all_records(td)
```

Delete DSF__Record

To Delete your `DSFRecord`:

```
>>> record.delete()
```

DSFRecordSet

class dyn.tm.services.dsf.**DSFRecordSet**(*rdata_class*, *label=None*, *ttl=None*, *automation=None*, *serve_count=None*, *fail_count=None*, *trouble_count=None*, *eligible=None*, *dsf_monitor_id=None*, *records=None*, ***kwargs*)

A Collection of DSFRecord Type objects belonging to a `DSFFailoverChain`

__init__(*rdata_class*, *label=None*, *ttl=None*, *automation=None*, *serve_count=None*, *fail_count=None*, *trouble_count=None*, *eligible=None*, *dsf_monitor_id=None*, *records=None*, ***kwargs*)

Create a new `DSFRecordSet` object

Parameters

- **rdata_class** – The type of rdata represented by this `DSFRecordSet`
- **label** – A unique label for this `DSFRecordSet`
- **ttl** – Default TTL for `DSFRecord`'s within this `DSFRecordSet`
- **automation** – Defines how eligible can be changed in response to monitoring
- **serve_count** – How many Records to serve out of this `DSFRecordSet`
- **fail_count** – The number of Records that must not be okay before this `DSFRecordSet` becomes ineligible.
- **trouble_count** – The number of Records that must not be okay before this `DSFRecordSet` becomes in trouble.
- **eligible** – Indicates whether or not this `DSFRecordSet` can be served.
- **dsf_monitor_id** – The unique system id of the DSF Monitor attached to this `DSFRecordSet`
- **records** – A list of `DSFRecord`'s within this `DSFRecordSet`
- **kwargs** – Used for manipulating additional data to be specified by the creation of other system objects.

add_to_failover_chain(*failover_chain*, *service=None*, *publish=True*, *notes=None*)

Creates and links this `DSFRecordSet` to the passed in `DSFFailoverChain` Object

Parameters

- **failover_chain** – Can either be the dsf_record_set_failover_chain_id or a `DSFFailoverChain` Object.
- **service** – Only necessary is rs_chain is passed in as a string. This can be a `TrafficDirector` Object. or the _service_id
- **publish** – Publish on execution (Default = True)
- **notes** – Optional Zone publish Notes

automation

Defines how eligible can be changed in response to monitoring

delete(*notes=None*, *publish=True*)
Delete this DSFRecordSet from the Dynect System :param notes: Optional zone publish notes :param publish: Publish at run time. Default is True

dsf_id
The unique system id of the TrafficDirector This DSFRecordSet is attached to

dsf_monitor_id
The unique system id of the DSF Monitor attached to this DSFRecordSet

eligible
Indicates whether or not this DSFRecordSet can be served

fail_count
The number of Records that must not be okay before this DSFRecordSet becomes ineligible.

implicitPublish
Toggle for this specific DSFRecordSet for turning on and off implicit Publishing for record Updates.

implicit_publish
Toggle for this specific DSFRecordSet for turning on and off implicit Publishing for record Updates.

label
A unique label for this DSFRecordSet

publish(*notes=None*)
Publish changes to TrafficDirector. :param notes: Optional Note that will be added to the zone notes of

zones attached to this service.

publish_note
Returns Current Publish Note, which will be used on the next publish action

rdata_class
The rdata property is a read-only attribute

record_set_id
The unique system id of this DSFRecordSet

records
The list of DSFRecord types that are stored in this DSFRecordSet

refresh()
Pulls data down from Dynect System and repopulates DSFRecordSet

serve_count
How many Records to serve out of this DSFRecordSet

set_monitor(*monitor*)
For attaching a DSFMonitor to this record_set :param monitor: a DSFMonitor or string of the dsf_monitor_id

to attach to this record_set

status
The current status of this DSFRecordSet

to_json(*svc_id=None*, *skip_svc=False*)
Convert this DSFRecordSet to a JSON blob

trouble_count
The number of Records that must not be okay before this DSFRecordSet becomes in trouble.

ttl
Default TTL for `DSFRecord`'s within this `DSFRecordSet`

DSFRecordSet Examples

The following examples highlight how to use the `DSFRecordSet` classes to get/create/update/delete `DSFRecordSet`'s on the dyn.tm System and how to edit these objects from within a Python script.

Create DSFRecordSet

We'll assume you already have a `DSFFailoverChain` object named `failover_chain` in existence for this example.

```
>>> from dyn.tm.services.dsf import DSFRecordSet
>>> #set up recordset for A records,
>>> record_set = DSFRecordSet('A', label='Record_set_test', ttl=60)
>>> #Now, we create this record_set by adding it to an existing failover_chain
>>> record_set.add_to_failover_chain(failover_chain) #This is automatically published.
```

To make the record_set and its child A records in one create action:

```
>>> from dyn.tm.services.dsf import DSFRecordSet, DSFARecord
>>> #Create A Record Prototypes
>>> record1 = DSFARecord('10.1.1.1', label='TEST RECORD 10', weight=1, automation=
↪'auto', eligible=True)
>>> record2 = DSFARecord('20.1.1.1', label='TEST RECORD 20', weight=1, automation=
↪'auto', eligible=True)
>>> #set up record_set for A records and pass in the two record protypes,
>>> record_set = DSFRecordSet('A', label='Record_set_test', ttl=60, records=[record1,␣
↪record2])
>>> #Now, we create this record_set by adding it to an existing failover_chain
>>> record_set.add_to_failover_chain(failover_chain) #This is automatically published.
```

As with all other DSF objects, the prototypes `record1` `record2` can't be used in CRUD operations. You must access these objects within the record_set.

```
>>> record_set.records
>>>[<ARecord>: 10.1.1.1, <ARecord>: 20.1.1.1]
```

Update DSFRecordSet

To change the label for the above `DSFRecordset`:

```
>>> record_set.label = 'New Name'  #This gets published implicitly
>>> #Check to see if it really changed.
>>> record_set.label.label
>>>'New Name'
```

Implicit publishing can be turned off for any object if that is undesirable, check `Modifying Traffic Director Service Properties` below for an example and explanation

Adding DSFMonitor to DSFRecordSet

To add a `DSFMonitor` to your `DSFRecordset`:

Existing `DSFRecordset`:

```
>>> from dyn.tm.services.dsf import DSFMonitor
>>> #create your monitor
>>> monitor = DSFMonitor('testmonitor', 'HTTP', 1, 60, 1, port=80)
>>> #or get an existing one (example)
>>> from dyn.tm.services.dsf import get_all_dsf_monitors
>>> monitor = get_all_dsf_monitors()[0]
>>> #Now attach monitor to record_set
>>> record_set.set_monitor(monitor)
```

New `DSFRecordset`:

```
>>> #Create or get your monitor object as above.
>>> record_set = DSFRecordSet('A', label='Record_set_test', ttl=60, dsf_monitor_
↪id=monitor.dsf_monitor_id)
>>> record_set.add_to_failover_chain(failover_chain) #create record_set
```

Get All DSFRecordSet

To get all `DSFRecordSet`: from a certain `TrafficDirector`:

```
>>> from dyn.tm.services.dsf import get_all_record_sets
>>> #Pass in a :class:`TrafficDirector`: instance to the following call:
>>> get_all_record_sets(td)
```

Delete DSFRecordSet

To Delete your `DSFRecordset`:

```
>>> record_set.delete()
```

This will delete all child records attached to this object!

DSFFailoverChain

class `dyn.tm.services.dsf.`**`DSFFailoverChain`**(*label=None*, *core=None*, *record_sets=None*, ***kwargs*)

docstring for DSFFailoverChain

`__init__`(*label=None*, *core=None*, *record_sets=None*, ***kwargs*)

Create a `DSFFailoverChain` object

Parameters

- **`label`** – A unique label for this `DSFFailoverChain`
- **`core`** – Indicates whether or not the contained `DSFRecordSets` are core record sets
- **`record_sets`** – A list of `DSFRecordSet`'s for this `DSFFailoverChain`

add_to_response_pool(*response_pool*, *service=None*, *publish=True*, *notes=None*)
Creates and Adds this DSFFailoverChain to a TrafficDirector service.

Parameters

- **response_pool** – Can either be the response_pool_id or a DSFResponsePool Object.
- **service** – Only necessary when response_pool is passed as a string. Can either be the service_id or a TrafficDirector
- **publish** – Publish on execution (Default = True)
- **notes** – Optional Zone publish Notes

core
Indicates whether or not the contained DSFRecordSet's are core record sets.

delete(*notes=None*, *publish=True*)
Delete this DSFFailoverChain from the Dynect System :param notes: Optional zone publish notes :param publish: Publish at run time. Default is True

dsf_id
The unique system id of the TrafficDirector This DSFFailoverChain is attached to

failover_chain_id
The unique system id of this DSFFailoverChain

implicitPublish
Toggle for this specific DSFFailoverChain for turning on and off implicit Publishing for record Updates.

implicit_publish
Toggle for this specific DSFFailoverChain for turning on and off implicit Publishing for record Updates.

label
A unique label for this DSFFailoverChain

publish(*notes=None*)
Publish changes to TrafficDirector. :param notes: Optional Note that will be added to the zone

notes of zones attached to this service.

publish_note
Returns Current Publish Note, which will be used on the next publish action

record_sets
A list of DSFRecordSet connected to this DSFFailvoerChain

refresh()
Pulls data down from Dynect System and repopulates DSFFailoverChain

response_pool_id
The unique system id of the DSFResponsePool this DSFFailoverChain is attached to

to_json(*svc_id=None*, *skip_svc=False*)
Convert this DSFFailoverChain to a JSON blob

DSFFailoverChain Examples

The following examples highlight how to use the `DSFFailoverChain` classes to get/create/update/delete `DSFFailoverChain`'s on the dyn.tm System and how to edit these objects from within a Python script.

Create DSFFailoverChain

We'll assume you already have a `DSFResponsePool` object named `response_pool` in existence for this example.

```
>>> from dyn.tm.services.dsf import DSFFailoverChain
>>> #set up failover_chain
>>> failover_chain = DSFFailoverChain(label='TEST Chain')
>>> #Now, we create this failover_chain by adding it to an existing response_pool
>>> failover_chain.add_to_response_pool(response_pool) #This is automatically␣
↪published.
```

To make the failover_chain and its child record_set in one create action:

```
>>> from dyn.tm.services.dsf import DSFFailoverChain, DSFRecordSet
>>> #set up record_set prototype
>>> record_set = DSFRecordSet('A', label='Record_set_test', ttl=60,)
>>> #set up failover_chain and pass in the record_set prototype
>>> failover_chain = DSFFailoverChain(label='TEST Chain', record_sets=[record_set])
>>> #Now, we create this failover_chain by adding it to an existing response_pool
>>> failover_chain.add_to_response_pool(response_pool) #This is automatically␣
↪published.
```

You can continue nesting beyond record_set by adding records = [record1...] to the record_set prototype. See `TrafficDirector` example for a larger example,

As with all other DSF objects, the prototypes record_set can't be used in CRUD operations. You must access these objects within the failover_chain.

```
>>> failover_chain.record_sets
>>>[<DSFRecordSet>: RDClass: A, Label: Record_set_test, ID: r6e1_IkchB-
↪Yp93rAEClo8QbZzA]
```

Update DSFFailoverChain

To change the label for the above `DSFFailoverChain`:

```
>>> failover_chain.label = 'New Name'  #This gets published implicitly
>>> #Check to see if it really changed.
>>> failover_chain.label
>>>'New Name'
```

Implicit publishing can be turned off for any object if that is undesirable, check `Modifying Traffic Director Service Properties` below for an example and explanation

Get All DSFFailoverChain

To get all `DSFFailoverChain`: from a certain `TrafficDirector`:

```
>>> from dyn.tm.services.dsf import get_all_failover_chains
>>> #Pass in a :class:`DSFFailoverChain`: instance to the following call:
>>> get_all_failover_chains(td)
```

Delete DSFFailoverChain

To Delete your `DSFFailoverChain`:

```
>>> failover_chain.delete()
```

This will delete all child records attached to this object!

DSFResponsePool

class `dyn.tm.services.dsf.`**`DSFResponsePool`**(*label*, *core_set_count=1*, *eligible=True*, *automation='auto'*, *dsf_ruleset_id=None*, *index=None*, *rs_chains=None*, ***kwargs*)

docstring for DSFResponsePool

`__init__`(*label*, *core_set_count=1*, *eligible=True*, *automation='auto'*, *dsf_ruleset_id=None*, *index=None*, *rs_chains=None*, ***kwargs*)

Create a `DSFResponsePool` object

Parameters

- **`label`** – A unique label for this `DSFResponsePool`
- **`core_set_count`** – If fewer than this number of core record sets are eligible, status will be set to fail
- **`eligible`** – Indicates whether or not the `DSFResponsePool` can be served
- **`automation`** – Defines how eligible can be changed in response to monitoring
- **`dsf_ruleset_id`** – Unique system id of the Ruleset this `DSFResponsePool` is attached to
- **`index`** – When specified with dsf_ruleset_id, indicates the position of the `DSFResponsePool`
- **`rs_chains`** – A list of `DSFFailoverChain` that are defined for this `DSFResponsePool`

`automation`

Defines how eligiblity can be changed in response to monitoring

`core_set_count`

If fewer than this number of core record sets are eligible, status will be set to fail

`create`(*service*, *publish=True*, *notes=None*)

Adds this `DSFResponsePool` to the passed in `TrafficDirector` :param service: a `TrafficDirector` or id string for the

`TrafficDirector` you wish to add this `DSFResponsePool` to.

Parameters

- **`publish`** – publish at execution time. Default = True

- **notes** – Optional Zone publish Notes

delete(*notes=None*, *publish=True*)
Delete this DSFResponsePool from the DynECT System :param notes: Optional zone publish notes :param publish: Publish at run time. Default is True

dsf_id
The unique system id of the TrafficDirector This DSFResponsePool is attached to

eligible
Indicates whether or not the DSFResponsePool can be served

failover_chains
A list of DSFFailoverChain that are defined for this DSFResponsePool

implicitPublish
Toggle for this specific DSFResponsePool for turning on and off implicit Publishing for record Updates.

implicit_publish
Toggle for this specific DSFResponsePool for turning on and off implicit Publishing for record Updates.

label
A unique label for this DSFResponsePool

publish(*notes=None*)
Publish changes to TrafficDirector. :param notes: Optional Note that will be added to the zone notes of zones attached to this service.

publish_note
Returns Current Publish Note, which will be used on the next publish action

refresh()
Pulls data down from Dynect System and repopulates DSFResponsePool

response_pool_id
The Unique system id of this DSFResponsePool

rs_chains
A list of DSFFailoverChain that are defined for this DSFResponsePool (legacy call)

ruleset_ids
List of Unique system ids of the DSFRuleset`s this :class:`DSFResponsePool is attached to

to_json(*svc_id=None*, *skip_svc=False*)
Convert this DSFResponsePool to a JSON blob

DSFResponsePool Examples

The following examples highlight how to use the DSFResponsePool classes to get/create/update/delete DSFResponsePool's on the dyn.tm System and how to edit these objects from within a Python script.

Create DSFResponsePool

Because the DSFReponsePool is at the bottom of the tree, there is nothing to attach to it except for the TrafficDirector service.

```
>>> from dyn.tm.services.dsf import DSFResponsePool
>>> #set up Response Pool with label
>>> response_pool = DSFResponsePool(label='TEST Pool')
>>> #Now, we create this response_pool by passing in the TrafficDirector object
>>> response_pool.create(td) #This is automatically published.
```

To make the response_pool and its child failover_chain in one create action:

```
>>> from dyn.tm.services.dsf import DSFFailoverChain, DSFResponsePool
>>> #set up failover_chain prototype
>>> failover_chain = DSFFailoverChain(label='TEST Chain')
>>> #set up response_pool and pass in the failover_chain prototype
>>> response_pool = DSFResponsePool(label='TEST Pool', rs_chains=[failover_chain])
>>> #Now, we create this response_pool by adding it to an existing TrafficDirector␣
↪service
>>> response_pool.create(td) #This is automatically published.
```

You can continue nesting beyond failover_chain by adding records_set = [record_set1...] to the failover_chain prototype. See `TrafficDirector` example for a larger example,

As with all other DSF objects, the prototypes failover_chain can't be used in CRUD operations. You must access these objects within the response_pool.

```
>>> response_pool.failover_chains
>>>[<DSFFailoverChain>: Label: TEST Chain, ID: AFUQpP2GRADINM1W12j_AVp_AX0]
```

Update DSFResponsePool

To change the label for the above `DSFResponsePool`:

```
>>> response_pool.label = 'New Name'  #This gets published implicitly
>>> #Check to see if it really changed.
>>> response_pool.label
>>>'New Name'
```

Implicit publishing can be turned off for any object if that is undesirable, check `Modifying Traffic Director Service Properties` below for an example and explanation

Get All DSFResponsePool

To get all `DSFResponsePool`: from a certain `TrafficDirector`:

```
>>> from dyn.tm.services.dsf import get_all_response_pools
>>> #Pass in a :class:`DSFResponsePool`: instance to the following call:
>>> get_all_response_pools(td)
```

Delete DSFResponsePool

To Delete your `DSFResponsePool`:

```
>>> response_pool.delete()
```

This will delete all child records attached to this object!

DSFRuleset

class dyn.tm.services.dsf.**DSFRuleset**(*label*, *criteria_type*, *response_pools*, *criteria=None*, *failover=None*, ***kwargs*)

docstring for DSFRuleset

__init__(*label*, *criteria_type*, *response_pools*, *criteria=None*, *failover=None*, ***kwargs*)

Create a DSFRuleset object

Parameters

- **label** – A unique label for this DSFRuleset
- **criteria_type** – A set of rules describing what traffic is applied to the DSFRuleset
- **criteria** – Varied depending on the specified criteria_type
- **failover** – IP address or Hostname for a last resort failover.
- **response_pools** – A list of DSFResponsePool's for this DSFRuleset

add_failover_ip(*ip*, *publish=True*)

Adds passed in DSFResponsePool to the end of this DSFRuleSet. This effectively creates a special new Record chain with a single IP. It can be accessed as a responce pool with label equal to the ip passed in.

Parameters **publish** – Publish on execution (Default = True)

add_response_pool(*response_pool*, *index=0*, *publish=True*)

Adds passed in DSFResponsePool to this DSFRuleSet. By default this adds it to the front of the list.

Parameters

- **response_pool** – Can either be the response_pool_id or a DSFResponsePool Object.
- **index** – where in the list of response pools to place this pool. 0 is the first position, 0 is the default.
- **publish** – Publish on execution (Default = True)

create(*service*, *index=None*, *publish=True*, *notes=None*)

Adds this DSFRuleset to the passed in TrafficDirector

Parameters

- **service** – a TrafficDirector or id string for the TrafficDirector you wish to add this DSFRuleset to.
- **index** – in what position to serve this ruleset. 0 = first.
- **publish** – publish at execution time. Default = True
- **notes** – Optional Zone publish Notes

criteria

The criteria rules, will be varied depending on the specified criteria_type

criteria_type

A set of rules describing what traffic is applied to the DSFRuleset

delete(*notes=None*, *publish=True*)
Remove this DSFRuleset from it's associated TrafficDirector Service :param notes: Optional zone publish notes :param publish: Publish at run time. Default is True

dsf_id
The unique system id of the TrafficDirector This DSFRuleset is attached to

implicitPublish
Toggle for this specific DSFRuleset for turning on and off implicit Publishing for record Updates.

implicit_publish
Toggle for this specific DSFRuleset for turning on and off implicit Publishing for record Updates.

label
A unique label for this DSFRuleset

order_response_pools(*pool_list*, *publish=True*)
For reordering the ruleset list. simply pass in a list of :class:'DSFResponcePool's in the order you wish them to failover.

Parameters

- **pool_list** – ordered list of DSFResponcePool
- **publish** – Publish on execution. default = True

publish(*notes=None*)
Publish changes to TrafficDirector. :param notes: Optional Note that will be added to the zone notes

of zones attached to this service.

publish_note
Returns Current Publish Note, which will be used on the next publish action

refresh()
Pulls data down from Dynect System and repopulates DSFRuleset

remove_response_pool(*response_pool*, *publish=True*)
Removes passed in DSFResponsePool from this DSFRuleSet

Parameters

- **response_pool** – Can either be the response_pool_id or a *DSFResponsePool* Object.
- **publish** – Publish on execution (Default = True)

response_pools
A list of DSFResponsePool's for this DSFRuleset

ruleset_id
The unique system id of this DSFRuleset

DSFRuleset Examples

The following examples highlight how to use the DSFRuleset classes to get/create/update/delete DSFRuleset's on the dyn.tm System and how to edit these objects from within a Python script.

Create DSFRuleset

The `DSFRuleset` contains zero or more Response Pools, and belongs to the `TrafficDirector` service

```
>>> from dyn.tm.services.dsf import DSFRuleset
>>> #Make an empty ruleset:
>>> ruleset = DSFRuleset('The Rules', criteria_type='always', response_pools=[])
```

To make the ruleset and its create-link a response_pool in one create action:

```
>>> from dyn.tm.services.dsf import DSFRuleset, DSFResponsePool
>>> response_pool = DSFResponsePool(label='TEST Pool')
>>> #Make an empty ruleset:
>>> ruleset = DSFRuleset('The Rules', criteria_type='always', response_
↪pools=[response_pool])
```

You can continue nesting beyond response_pool by adding rs_chain = [failover_chain1...] to the response_pool prototype. See `TrafficDirector` example for a larger example,

As with all other DSF objects, the prototypes response_pool can't be used in CRUD operations. You must access these objects within the ruleset.

```
>>> ruleset.response_pools
>>>[<DSFResponsePool>: Label: TEST Pool, ID: NXAdxSrodSCUO_p9vbbpKuXJIOw]
```

Adding/Deleting/Modifying DSFResponsePools to DSFRuleset

The order of :class:'DSFResponsePool's is important in rulesets, so we have a number of functions for handling this. For this example assume we have 4 response pools pre-existing.

```
>>> #Lets add all 4 Response Pools to the ruleset.
>>> ruleset.add_response_pool(pool1) #First Pool
>>> ruleset.add_response_pool(pool2) #added to the front of the list
>>> ruleset.add_response_pool(pool3) #added to the front of the list
>>> #If we want pool4 to be at the back of the list we can specify the index.
>>> ruleset.add_response_pool(pool4, index=3)
>>> ruleset.response_pools
>>>[<DSFResponsePool>: Label: pool3, ID: 4Vu7lCEb3iDuATWq5Q6-5P-RAfU,
...<DSFResponsePool>: Label: pool2, ID: LPDIZfbr0gEVg-AR31CNE_wVDIg,
...<DSFResponsePool>: Label: pool1, ID: JybChuDQtCWSyADLFfqp2JKFYoE,
...<DSFResponsePool>: Label: pool4, ID: 3a-eVZYaRt3NeNxUXyA87OroswQ]
```

If you need to re-order your list, there is a helper function

```
>>> ruleset.order_response_pools([pool1,pool2,pool3,pool4])
>>> ruleset.response_pools
>>> [<DSFResponsePool>: Label: pool1, ID: JybChuDQtCWSyADLFfqp2JKFYoE,
...<DSFResponsePool>: Label: pool2, ID: LPDIZfbr0gEVg-AR31CNE_wVDIg,
...<DSFResponsePool>: Label: pool3, ID: 4Vu7lCEb3iDuATWq5Q6-5P-RAfU,
...<DSFResponsePool>: Label: pool4, ID: 3a-eVZYaRt3NeNxUXyA87OroswQ]
```

And, if you need to Delete a `DSFResponsePool` from the ruleset

```
>>> ruleset.remove_response_pool(pool3)
>>>[<DSFResponsePool>: Label: pool1, ID: JybChuDQtCWSyADLFfqp2JKFYoE,
```

```
...<DSFResponsePool>: Label: pool2, ID: LPDIZfbr0gEVg-AR31CNE_wVDIg,
...<DSFResponsePool>: Label: pool4, ID: 3a-eVZYaRt3NeNxUXyA87OroswQ]
```

Adding/manipulating a failover IP to DSFRuleset

`DSFRulesets` have the option to failover to a static IP. Behind the scenes, this is essential a full ResponsePool to Record chain with one single host or IP. when manipulating this value, keep that in mind.

Assume we have the same service as the Adding/Deleting/Modifying DSFResponsePools to DSFRuleset example.

```
>>> #To Add the failover IP.
>>> ruleset.add_failover_ip('1.2.3.4')
>>> # Notice how its essentially a Response_pool -> Record chain -- this is always␣
↪added to the end of the response pool list.
>>> ruleset.response_pools
>>>[<DSFResponsePool>: Label: pool1, ID: JybChuDQtCWSyADLFfqp2JKFYoE,
...<DSFResponsePool>: Label: pool2, ID: LPDIZfbr0gEVg-AR31CNE_wVDIg,
...<DSFResponsePool>: Label: pool4, ID: 3a-eVZYaRt3NeNxUXyA87OroswQ,
...<DSFResponsePool>: Label: 1.2.3.4, ID: wyUslh6c9eTXFvu7OSfW7S6Hj9I]
>>> # To modify the IP:
>>> ruleset.response_pools[3].rs_chains[0].record_sets[0].records[0].address = '10.10.
↪10.10'
>>> #The labels for the chain will still say 1.2.3.4, but the served records will be␣
↪10.10.10.10
```

Update DSFRuleset

To change the label for the above `DSFRuleset`:

```
>>> ruleset.label = 'New Name'  #This gets published implicitly
>>> #Check to see if it really changed.
>>> ruleset.label
>>>'New Name'
```

Get All DSFRuleset

To get all `DSFRuleset`: from a certain `TrafficDirector`:

```
>>> from dyn.tm.services.dsf import get_all_rulesets
>>> #Pass in a :class:`DSFRuleset`: instance to the following call:
>>> get_all_rulesets(td)
```

Delete DSFRuleset

To Delete your `DSFRuleset`:

```
>>> ruleset.delete()
```

This will NOT delete child records, however any child response pools and children that are not in other `DSFRuleset`s may not be displayed in the :class:`TrafficDirector` object as it builds its trees from the Rulesets. see `Traffic Director SDK Caveats`

DSFMonitor

class dyn.tm.services.dsf.**DSFMonitor**(**args*, ***kwargs*)
A Monitor for a TrafficDirector Service

__init__(**args*, ***kwargs*)
Create a new DSFMonitor object

Parameters

- **label** – A unique label to identify this DSFMonitor
- **protocol** – The protocol to monitor. Must be one of 'HTTP', 'HTTPS', 'PING', 'SMTP', or 'TCP'
- **response_count** – The number of responses to determine whether or not the endpoint is 'up' or 'down'
- **probe_interval** – How often to run this DSFMonitor
- **retries** – How many retries this DSFMonitor should attempt on failure before giving up.
- **active** – Indicates if this DSFMonitor is active
- **options** – Additional options pertaining to this DSFMonitor
- **endpoints** – A List of DSFMonitorEndpoint's that are associated with this DSFMonitor

active
Returns whether or not this DSFMonitor is active. Will return either 'Y' or 'N'

delete()
Delete an existing DSFMonitor from the DynECT System

dsf_monitor_id
The unique system id of this DSFMonitor

endpoints
A list of the endpoints (and their statuses) that this DSFMonitor is currently monitoring.

label
A unique label to identify this DSFMonitor

options
Additional options pertaining to this DSFMonitor

probe_interval
How often to run this DSFMonitor

protocol
The protocol to monitor. Must be one of 'HTTP', 'HTTPS', 'PING', 'SMTP', or 'TCP'

response_count
The minimum number of agents reporting the host as up for failover not to occur. Must be 0, 1 or 2

retries
How many retries this DSFMonitor should attempt on failure before giving up.

DSFMonitor Examples

The following examples highlight how to use the DSFMonitor classes to get/create/update/delete DSFMonitor's on the dyn.tm System and how to edit these objects from within a Python script.

Create DSFMonitor

Unlike most of the other DSF objects, DSFMonitor publishes when the object is created.

```
>>> from dyn.tm.services.dsf import DSFMonitor
>>> monitor = DSFMonitor('MonitorLabel', 'HTTP', 1, 60, 1, port=8080)
>>> monitor.dsf_monitor_id
>>> u'SE-6GKx_tEBHyL4G_-i28R2QiNs'
```

Update DSFMonitor

To change the label for the above DSFRuleset:

```
>>> monitor.label = 'NewMonitorName'  #Changes are immediate
>>> #Check to see if it really changed.
>>> monitor.label
>>>'NewMonitorName'
```

Add To DSFMonitor to DSFRecordSet

See DSFRecordSet example.

Get All DSFMonitor

To get all DSFMonitor:

```
>>> from dyn.tm.services.dsf import get_all_dsf_monitors
>>> #Not a child class, monitors are their own entity, so no need to pass in a␣
↪:class:`TrafficDirector`:
>>> get_all_dsf_monitors()
```

Delete DSFMonitor

To Delete your DSFMonitor:

```
>>> monitor.delete()
```

DSFNotifier

class dyn.tm.services.dsf.**DSFNotifier**(**args*, ***kwargs*)

__init__(**args*, ***kwargs*)
Create a `Notifier` object

Parameters

- **label** –
- **recipients** – `list` of Contact Names
- **dsf_services** –
- **monitor_services** –

add_recipient(*new_recipient*, *format='email'*)

del_recipient(*recipient*)

delete()
Delete this `DSFNotifier` from the Dynect System

dsf_service_ids

label

link_id
Link ID connecting thie Notifier to TD service

monitor_service_ids

recipients

to_json()

DSFNotifier Examples

The following examples highlight how to use the `DSFNotifier` classes to get/create/update/delete `DSFNotifier`'s on the dyn.tm System and how to edit these objects from within a Python script.

Create DSFNotifier

Unlike most of the other DSF objects, `DSFNotifier` publishes when the object is created.

```
>>> from dyn.tm.services.dsf import DSFNotifier
>>> #When passing in recipients, pass in a list of strings(s) of your contact(s)␣
↪nickname(s)
>>> notifier = DSFNotifier('Notifier', recipients=['youruser'])
>>> notifier.dsf_notifier_id
>>> u'BHyL4GxatEBHyR2QiNT28R2QiNs'
```

You can add the new notifier directly to a `TrafficDirector` as well

```
>>> from dyn.tm.services.dsf import DSFNotifier
>>> #When passing in recipients, pass in a list of strings(s) of your contact(s)␣
↪nickname(s)
>>> notifier = DSFNotifier('Notifier', recipients=['youruser'], dsf_services=[td.
↪service_id])
>>> notifier.dsf_notifier_id
>>> u'xatEBHyQiNT28R2QiyR2QiNt28R'
```

Update DSFNotifier

To change the label for the above `DSFRuleset`:

```
>>> notifier.label = 'NewNotifierName'  #Changes are immediate
>>> #Check to see if it really changed.
>>> notifier.label
>>>'NewNotifierName'
```

Get All DSFNotifier

To get all `DSFNotifier`:

```
>>> from dyn.tm.services.dsf import get_all_dsf_notifiers
>>> #Not a child class, notifiers are their own entity, so no need to pass in a␣
↪:class:`TrafficDirector`:
>>> get_all_dsf_notifiers()
```

Delete DSFNotifier

To Delete your `DSFNotifier`:

```
>>> notifier.delete()
```

Traffic Director

class dyn.tm.services.dsf.**TrafficDirector**(**args*, ***kwargs*)

Traffic Director is a DNS based traffic routing and load balancing service that is Geolocation aware and can support failover by monitoring endpoints.

__init__(**args*, ***kwargs*)

Create a `TrafficDirector` object

Parameters

- **label** – A unique label for this `TrafficDirector` service
- **ttl** – The default TTL to be used across this service
- **publish** – If Y, service will be published on creation
- **notes** – Optional Publish Zone Notes.
- **nodes** – A Node Object, a zone, FQDN pair in a hash, or a list containing those two things (can be mixed) that are to be linked to this `TrafficDirector` service:
- **notifiers** – A list of notifier ids associated with this `TrafficDirector` service
- **rulesets** – A list of `DSFRulesets` that are defined for this `TrafficDirector` service

add_node(*node*)

A `DSFNode` object, or a zone, FQDN pair in a hash to be added to this `TrafficDirector` service

add_notifier(*notifier*, *notes=None*)
Links the DSFNotifier with the specified id to this Traffic Director service, Accepts DSFNotifier or Notifier or the notifier public id.

all_failover_chains
Returns All DSFFailoverChain in TrafficDirector

all_record_sets
Returns All DSFRecordSet in TrafficDirector

all_records
Returns All DSFRecord in TrafficDirector

all_response_pools
Returns All DSFResponsePool in TrafficDirector

all_rulesets
Returns All DSFRuleset in TrafficDirector

del_notifier(*notifier*, *notes=None*)
delinks the DSFNotifier with the specified id to this Traffic Director service. Accepts DSFNotifier or Notifier.

delete()
Delete this TrafficDirector from the DynECT System :param notes: Optional zone publish notes

failover_chains
A list of this TrafficDirector Services DSFFailoverChain's

implicitPublish
Toggle for this specific TrafficDirector for turning on and off implicit Publishing for record Updates.

implicit_publish
Toggle for this specific TrafficDirector for turning on and off implicit Publishing for record Updates.

label
A unique label for this TrafficDirector service

nodeObjects
A list of DSFNode Objects that are linked to this TrafficDirector service

node_objects
A list of DSFNode Objects that are linked to this TrafficDirector service

nodes
A list of hashes of zones, fqdn for each DSF node that is linked to this TrafficDirector service

notifiers
A list of names of DSFNotifier associated with this TrafficDirector service

order_rulesets(*ruleset_list*, *publish=True*)
For reordering the ruleset list. simply pass in a list of :class:'DSFRulesets's in the order you wish them to be served.

Parameters

- **ruleset_list** – ordered list of DSFRulesets
- **publish** – Publish on execution. default = True

publish(*notes=None*)
Publish changes to `TrafficDirector`. :param notes: Optional Note that will be added to the zone notes of zones attached to this service.

publish_note
Returns Current Publish Note, which will be used on the next publish action

record_sets
A list of this `TrafficDirector` Services `DSFRecordSet`'s

records
A list of this `TrafficDirector` Services' DSFRecords

refresh()
Pulls data down from Dynect System and repopulates `TrafficDirector`

remove_node(*node*)
A `DSFNode` object, or a zone, FQDN pair in a hash to be removed to this `TrafficDirector` service

remove_orphans()
Remove Record Set Chains which are no longer referenced by a `DSFResponsePool`

replace_all_rulesets(*rulesets*)
This request will replace all rulesets with a new list of rulesets.

Parameters **rulesets** – a list of rulesets :class:DSFRuleset to be published

to the service Warning! This call takes extra time as it is several api calls.

replace_one_ruleset(*ruleset*)
This request will replace a single ruleset and maintain the order of the list.

Warning! This call takes extra time as it is several api calls.

Parameters **ruleset** – A single object of :class:DSFRuleset' The replacement

is keyed by the DSFRuleset label value.

response_pools
A list of this `TrafficDirector` Services `DSFResponsePool`'s

revert_changes()
Clears the changeset for this service and reverts all non-published changes to their original state

rulesets
A list of `DSFRulesets` that are defined for this `TrafficDirector` service

service_id
The unique System id of this DSF Service

ttl
The default TTL to be used across this service

Traffic Director Examples

The following examples highlight how to use the `TrafficDirector` class to get/create/update/delete `TrafficDirector`'s on the dyn.tm System and how to edit these objects from within a Python script.

Creating an empty Traffic Director Service

The following shows the creation of the very most basic empty `TrafficDirector`

```
>>> from dyn.tm.services.dsf import TrafficDirector
>>> td = TrafficDirector('TD_test_1', rulesets=[])
>>> #Now, lets look at the ID to make sure it was actually created.
>>> td.service_id
>>>u'w8WWsaqJicADC8OD1k_3GSFru7M'
>>> #service_id will be a long hash
```

Adding a Ruleset to your Traffic Director Service

The TrafficDirector service has a cascading style of adding sub objects where the child object is added to the parent by either and `add_to_` function, or a create. This helps enforce that children objects do not become orphaned.

```
>>> #Continuing from the example above.
>>> from dyn.tm.services.dsf import DSFRuleset
>>> #Let's make a ruleset called 'The Rules' which always serves, and has no response_
↪pools
>>> ruleset = DSFRuleset('The Rules', criteria_type='always', response_pools=[])
>>> #Now, lets add that ruleset to the Traffic Director instance from above.
>>> ruleset.create(td)
>>> #Now, Verify it was added. The 'rulesets' getter will return a list of rulesets␣
↪attached to the td service instance.
>>> td.rulesets
>>>[<DSFRuleSet>: Label: The Rules, ID: gthPTkFOYUrJFymEknoHeezBeSQ]
```

Adding RecordSets, FailoverChains, RecordSets, and Records to your Traffic Director Service

Please see individual sections for instructions on how to actually do this, as with :class:‘DSFRuleset‘s, there is a cascading system:

TD <- Ruleset -> ResponsePool <- FailoverChain <- RecordSet <- Record

to ‘create’ each, the function looks like:

```
>>> ruleset.create(td)
>>> ruleset.add_response_pool(pool)
>>> pool.create(td)
>>> failoverchain.add_to_response_pool(pool)
>>> recordset.add_to_failover_chain(failoverchain)
>>> record.add_to_record_set(recordset)
```

Modifying Traffic Director Service Properties

You can modify such things as labels, ttl, etc for the `TrafficDirector` object. Note, that modifying these values will immediately publish them. This can be turned off as in the example below.

```
>>> #Continuing from the example above.
>>> #parameter updates will publish implicitly.
>>> td.label #check what the label is.
>>>u'TD_test_1'
>>> td.label='TD_test_2'
>>> td.label
>>>u'TD_test_2'
```

```
>>> #Now, say you don't want your update changes to be implicitly published. you can␣
↪turn off implicit publishing for
>>> #the service level changes.
>>> #!!!WARNING!!!! changing the implicit publish flag ONLY disables implicit␣
↪publishing for this Object,
>>> #not any of its children objects like Rulesets etc.
>>>
>>> td.label
>>>u'TD_test_2'
>>> td.implicitPublish = False
>>> td.label = 'TD_test_3'
>>> td.refresh() #pulls down fresh data from the system, as your td.label is now␣
↪stale due to it not being published
>>> td.label
>>>u'TD_test_2'
>>> td.ttl = 299
>>> td.refresh()
>>> td.ttl
>>>300
>>> td.publish()
>>> td.ttl
>>>299
>>> td.label
>>>u'TD_Test_3'
```

Getting an Existing Traffic Director Service

The following example shows how to get an existing `TrafficDirector` from the dyn.tm System

```
>>> # Continuing from the previous example
>>> id = td.service_id
>>> gotTD = TrafficDirector(id)
>>> gotTD.label
>>>u'TD_Test_3'
```

What if you don't know your service_id? But maybe you know the name...

```
>>> from dyn.tm.services.dsf import get_all_dsf_services
>>> get_all_dsf_services()
>>>[<TrafficDirector>: notme, ID: qzoiassV-quZ_jGh7jbn_PfYNxY,
...<TrafficDirector>: notmeeither, ID: qdE-zi4k7zEVhH6jWugVSbiIxdA,
...<TrafficDirector>: imtheone, ID: AwqcnhOZ6r1aCpIZFIj4mTwdd9Y]
>>> myTD = get_all_dsf_services()[2]
>>> myTD.label
>>>u'imtheone'
```

Adding/Deleting a Notifier to your Traffic Director Service

You can add notifiers to your Traffic Director service in the following ways:

Example 1:

```
>>> from dyn.tm.services.dsf import DSFNotifier
>>> notifier = DSFNotifier('deleteme', recipients=['youruser'])
>>> td.add_notifier(notifier1)
```

```
>>> td.refresh()
>>> td.notifiers
>>>[<DSFNotifier>: deleteme, ID: 8lJ9LUcP9sIuB8V58zsGWVu1Hys]
>>> #To delete:
>>> td.del_notifier(notifier1)
>>> td.refresh()
>>> td.notifiers
>>>[]
```

Example 2:

```
>>> #Notifiers can also be added at the creation time of the Notifier by␣
↪passing in the service_id
>>> from dyn.tm.services.dsf import DSFNotifier
>>> notifier = DSFNotifier('deleteme', recipients=['youruser'], dsf_
↪services=[td.service_id])
>>> td.refresh()
>>> td.notifiers
>>>[<DSFNotifier>: deleteme, ID: q-hZOVTn2Q_VCX1LFMSI-4LPTww]
```

Deleting Traffic Director Service

You can also delete your service in the following manner:

>>>td.delete() >>>td.refresh() >>>DynectGetError: detail: No service found.

Creating a fully populated Traffic Director Service

The following example shows how to create a new `TrafficDirector` on the dyn.tm System and how to edit some of the fields using the returned `TrafficDirector` object.

```
>>> #A fully populated service can achieved by creating a full chain and passing␣
↪child objects into each parent object.
>>> #These objects are effectively constructor objects. In other words, they will be␣
↪useless for CRUD operations, except for
>>> #The TrafficDirector object. There are other means for achieving CRUD operations␣
↪as you will see.
>>>
>>> from dyn.tm.services.dsf import *
>>> from dyn.tm.zones import Node
>>>
>>> #Lets start with our objects that are actually created when the command is␣
↪executed.
>>>
>>> #First, lets make our Monitor. we pass this in to the recordset later. This␣
↪monitor is created at execution time.
>>> monitor = DSFMonitor('MonLabel', 'HTTP', 1, 60, 1, port=8080)
>>>
>>> #Second, lets make a new Notifier -- this is optional. We'll assume you have a␣
↪contact named 'contactname'
>>> notifier = DSFNotifier('Notifier', recipients=['contactname'])
>>>
>>>
>>> #Next lets make our A Record prototype:
>>> a_rec = DSFARecord('1.1.1.1', ttl=60, label='RecordLabel')
```

```
>>>
>>> #Next, lets create the record_set. Note how we pass in the a_rec A Record Object,␣
↪and the monitor_id
>>> record_set = DSFRecordSet('A', label='RSLabel', dsf_monitor_id = monitor.dsf_
↪monitor_id,records=[a_rec])
>>>
>>> #Next, lets create the failover chain Note how we pass in the record_set␣
↪RecordSet Object
>>> failover_chain = DSFFailoverChain(label='FCLabel', record_sets=[record_set])
>>>
>>> #Next, lets create the response pool Note how we pass in the failover_chain␣
↪Failover Chain Object
>>> rp = DSFResponsePool(label='RPLabel', rs_chains=[failover_chain])
>>> criteria = {'geoip': {'country': ['US']}}
>>>
>>> #Next, lets create the ruleset Note how we pass in the rp Response Pool Object
>>> ruleset = DSFRuleset(label='RSLabel', criteria_type='geoip',
...                      criteria=criteria, response_pools=[rp])
>>>
>>> #Now, lets create a Node object. This is used for attaching the service to a Node␣
↪(or zone)
>>> node = Node('example.com',fqdn = 'example.com.')
>>>
>>> #Finally, we pass all of this in. upon command execution the service will have␣
↪been created.
>>>
>>> dsf = TrafficDirector('Test_Service', rulesets=[ruleset], nodes=[node],␣
↪notifiers=[notifier])
```

Now that you have created your service in one fell swoop, there are a few things you must know:

Prototype objects like your DSFARecord, DSFRecordSet are just that, prototypes. You can't perform CRUD operations on them. This goes for any child object where you pass in prototypes. See examples below:

```
>>> #Trying to access a prototype
>>> a_rec.address='1.2.3.4'
>>>DynectUpdateError: record_update: No service found.
```

```
>>> #Instead, do this:
>>> dsf.records
>>> [<ARecord>: 1.1.1.1]
>>> dsf.records[0].address='1.2.3.4'
>>> dsf.records[0].address
>>> u'1.2.3.4'
```

Traffic Director SDK Caveats

- Creating a fully populated service with prototypes leaves the prototypes unusable. CRUD capabilities can only be achieved by accessing data within the `TrafficDirector` object. Accessors are `records`, `record_sets`, `failover_chains`, `response_pools`, `rulesets`
- Accessors like in the previous bullet point only work if the object is fully linked to the service. In other words, you can have a full response_pool, but if it does not belong to a ruleset, then it will not show up. To list all objects under the service, including orphands you must use `all_records`, `all_record_sets`, `all_failover_chains`, `all_response_pools`, `all_rulesets`

- Some `records`, `record_sets`, `failover_chains`, `response_pools`, `rulesets` will appear multiple times. This is because these record trees are built from the ruleset, and if one response pool belongs to multiple Rulesets, then its children will appear as many times as is exists as a ruleset member.
- `refresh()` is your friend. When modifying child objects from a parent sometimes the parent doesn't know about the changes. If you do a `refresh()` on the `TrafficDirector` object it will pull down the latest data from the Dynect System.
- `publish()` is run on the `TrafficDirector` as a whole, even when run from a child object.
- `implicitPublish` is non cascading. It is locally bound to the specific object, or child object.

GSLB

The `services` module contains interfaces to all of the various service management features offered by the dyn.tm REST API

Monitor

class `dyn.tm.services.gslb.`**`Monitor`**(*protocol*, *interval*, *retries=None*, *timeout=None*, *port=None*, *path=None*, *host=None*, *header=None*, *expected=None*)

A `Monitor` for a GSLB Service

`__init__`(*protocol*, *interval*, *retries=None*, *timeout=None*, *port=None*, *path=None*, *host=None*, *header=None*, *expected=None*)

Create a `Monitor` object

Parameters

- **`protocol`** – The protocol to monitor. Must be either HTTP, HTTPS, PING, SMTP, or TCP
- **`interval`** – How often (in minutes) to run the monitor. Must be 1, 5, 10, or 15,
- **`retries`** – The number of retries the monitor attempts on failure before giving up
- **`timeout`** – The amount of time in seconds before the connection attempt times out
- **`port`** – For HTTP(S)/SMTP/TCP probes, an alternate connection port
- **`path`** – For HTTP(S) probes, a specific path to request
- **`host`** – For HTTP(S) probes, a value to pass in to the Host
- **`header`** – For HTTP(S) probes, additional header fields/values to pass in, separated by a newline character.
- **`expected`** – For HTTP(S) probes, a string to search for in the response. For SMTP probes, a string to compare the banner against. Failure to find this string means the monitor will report a down status.

`expected`

For HTTP(S) probes, a string to search for in the response. For SMTP probes, a string to compare the banner against. Failure to find this string means the monitor will report a down status

`header`

For HTTP(S) probes, additional header fields/values to pass in, separated by a newline character

`host`

For HTTP(S) probes, a value to pass in to the Host

interval
How often to run this monitor

path
For HTTP(S) probes, a specific path to request

port
For HTTP(S)/SMTP/TCP probes, an alternate connection port

protocol
The protocol to monitor

retries
The number of retries the monitor attempts on failure before giving up

status
Get the current status of this `HealthMonitor` from the DynECT System

timeout
The amount of time in seconds before the connection attempt times out

to_json()
Convert this `HealthMonitor` object to a JSON blob

GSLBRegionPoolEntry

class dyn.tm.services.gslb.**GSLBRegionPoolEntry**(*zone*, *fqdn*, *region_code*, *address*, **args*, ***kwargs*)
`GSLBRegionPoolEntry`

__init__(*zone*, *fqdn*, *region_code*, *address*, **args*, ***kwargs*)
Create a `GSLBRegionPoolEntry` object

Parameters

- **zone** – Zone monitored by this `GSLBRegionPoolEntry`
- **fqdn** – The fqdn of the specific node which will be monitored by this `GSLBRegionPoolEntry`
- **region_code** – ISO Region Code for this `GSLBRegionPoolEntry`
- **address** – The IP address or FQDN of this Node IP
- **label** – Identifying descriptive information for this `GSLBRegionPoolEntry`
- **weight** – A number in the range of 1-14 controlling the order in which this `GSLBRegionPoolEntry` will be served
- **serve_mode** – Sets the behavior of this particular record. Must be one of 'always', 'obey', 'remove', 'no'

address
The IP address or FQDN of this Node IP

delete()
Delete this `GSLBRegionPoolEntry` from the DynECT System

fqdn
The fqdn of the specific node which will be monitored by this `GSLBRegionPoolEntry`

label
Identifying descriptive information for this `GSLBRegionPoolEntry`

region_code
ISO Region Code for this `GSLBRegionPoolEntry`

serve_mode
Sets the behavior of this particular record. Must be one of 'always', 'obey', 'remove', or 'no'

sync()
Sync this `GSLBRegionPoolEntry` object with the DynECT System

task
`Task` for most recent system action on this `ActiveFailover`.

to_json()
Convert this object into a json blob

weight
A number in the range of 1-14 controlling the order in which this `GSLBRegionPoolEntry` will be served.

zone
Zone monitored by this `GSLBRegionPoolEntry`

GSLBRegion

class dyn.tm.services.gslb.**GSLBRegion**(*zone*, *fqdn*, *region_code*, **args*, ***kwargs*)
docstring for GSLBRegion

__init__(*zone*, *fqdn*, *region_code*, **args*, ***kwargs*)
Create a `GSLBRegion` object

Parameters

- **zone** – Zone monitored by this `GSLBRegion`
- **fqdn** – The fqdn of the specific node which will be monitored by this `GSLBRegion`
- **region_code** – ISO region code of this `GSLBRegion`
- **pool** – (*arg) The IP Pool list for this `GSLBRegion`
- **serve_count** – How many records will be returned in each DNS response
- **failover_mode** – How the `GSLBRegion` should failover. Must be one of 'ip', 'cname', 'region', 'global'
- **failover_data** – Dependent upon failover_mode. Must be one of 'ip', 'cname', 'region', 'global'

delete()
Delete this `GSLBRegion`

failover_data
Dependent upon failover_mode. Must be one of 'ip', 'cname', 'region', 'global'

failover_mode
How the `GSLBRegion` should failover. Must be one of 'ip', 'cname', 'region', 'global'

fqdn
The fqdn of the specific node which will be monitored by this `GSLBRegion`

pool
The IP Pool list for this `GSLBRegion`

region_code
ISO region code of this GSLBRegion

serve_count
How many records will be returned in each DNS response

sync()
Sync this GSLBRegion object with the DynECT System

task
Task for most recent system action on this ActiveFailover.

zone
Zone monitored by this GSLBRegion

GSLB

class dyn.tm.services.gslb.**GSLB**(*zone*, *fqdn*, **args*, ***kwargs*)
A Global Server Load Balancing (GSLB) service

__init__(*zone*, *fqdn*, **args*, ***kwargs*)
Create a GSLB object

Parameters

- **auto_recover** – Indicates whether or not the service should automatically come out of failover when the IP addresses resume active status or if the service should remain in failover until manually reset. Must be 'Y' or 'N'
- **ttl** – Time To Live in seconds of records in the service. Must be less than 1/2 of the Health Probe's monitoring interval. Must be one of 30, 60, 150, 300, or 450
- **notify_events** – A comma separated list of the events which trigger notifications. Must be one of 'ip', 'svc', or 'nosrv'
- **syslog_server** – The Hostname or IP address of a server to receive syslog notifications on monitoring events
- **syslog_port** – The port where the remote syslog server listens for notifications
- **syslog_ident** – The ident to use when sending syslog notifications
- **syslog_facility** – The syslog facility to use when sending syslog notifications. Must be one of 'kern', 'user', 'mail', 'daemon', 'auth', 'syslog', 'lpr', 'news', 'uucp', 'cron', 'authpriv', 'ftp', 'ntp', 'security', 'console', 'local0', 'local1', 'local2', 'local3', 'local4', 'local5', 'local6', or 'local7'
- **syslog_delivery** – The syslog delivery action type. 'all' will deliver notifications no matter what the endpoint state. 'change' (default) will deliver only on change in the detected endpoint state
- **region** – A list of GSLBRegion's
- **monitor** – The health Monitor for this service
- **contact_nickname** – Name of contact to receive notifications
- **syslog_probe_fmt** – see below for format:
- **syslog_status_fmt** – see below for format: Use the following format for syslog_xxxx_fmt paramaters. %hos hostname %tim current timestamp or monitored interval %reg region code %sta status %ser record serial %rda rdata %sit monitoring

site %rti response time %msg message from monitoring %adr address of monitored node %med median value %rts response times (RTTM)

- **recovery_delay** – number of up status polling intervals to consider service up

activate()
Activate this GSLB service on the DynECT System

active
Indicates if the service is active. When setting directly, rather than using activate/deactivate valid arguments are 'Y' or True to activate, or 'N' or False to deactivate. Note: If your service is already active and you try to activate it, nothing will happen. And vice versa for deactivation.

Returns An Active object representing the current state of this GSLB Service

auto_recover
Indicates whether or not the service should automatically come out of failover when the IP addresses resume active status or if the service should remain in failover until manually reset. Must be 'Y' or 'N'

contact_nickname
Name of contact to receive notifications from this GSLB service

deactivate()
Deactivate this GSLB service on the DynECT System

delete()
Delete this GSLB service from the DynECT System

monitor
The health Monitor for this service

notify_events
A comma separated list of the events which trigger notifications. Must be one of 'ip', 'svc', or 'nosrv'

recover(*address=None*)
Recover the GSLB service on the designated zone node or a specific node IP within the service

recovery_delay

region
A list of GSLBRegion's

status
The current state of the service. Will be one of 'unk', 'ok', 'trouble', or 'failover'

sync()
Sync this GSLB object with the DynECT System

syslog_delivery

syslog_facility
The syslog facility to use when sending syslog notifications. Must be one of 'kern', 'user', 'mail', 'daemon', 'auth', 'syslog', 'lpr', 'news', 'uucp', 'cron', 'authpriv', 'ftp', 'ntp', 'security', 'console', 'local0', 'local1', 'local2', 'local3', 'local4', 'local5', 'local6', or 'local7'

syslog_ident
The ident to use when sending syslog notifications

syslog_port
The port where the remote syslog server listens for notifications

syslog_probe_format

syslog_server
The Hostname or IP address of a server to receive syslog notifications on monitoring events

syslog_status_format

task
`Task` for most recent system action on this `GSLB`.

ttl
Time To Live in seconds of records in the service. Must be less than 1/2 of the Health Probe's monitoring interval. Must be one of 30, 60, 150, 300, or 450

GSLB Examples

The following examples highlight how to use the `GSLB` class to get/create `GSLB`'s on the dyn.tm System and how to edit these objects from within a Python script.

Creating a new GSLB Service

The following example shows how to create a new `GSLB` on the dyn.tm System and how to edit some of the fields using the returned `GSLB` object.

```
>>> from dyn.tm.services.gslb import Monitor, GSLBRegionPoolEntry, \
...     GSLBRegion, GSLB
>>> # Create a dyn.tmSession
>>> # Assuming you own the zone 'example.com'
>>> zone = 'example.com'
>>> fqdn = zone + '.'
>>> pool = GSLBRegionPoolEntry(zone, fqdn, 'global', '8.8.4.4', None,
...                            label='APIv2 GSLB')
>>> region = GSLBRegion(zone, fqdn, 'mycontactnickname', pool=[pool])
>>> monitor = Monitor('HTTP', 5, expected='Example')
>>> gslb = GSLB(zone, fqdn, 'mycontactname', region=[region], monitor=monitor)
```

Getting an Existing GSLB Service

The following example shows how to get an existing `GSLB` from the dyn.tm System and how to edit some of the same fields mentioned above.

```
>>> from dyn.tm.services.gslb import GSLB
>>> # Create a dyn.tmSession
>>> # Once again, assuming you own 'example.com'
>>> zone = 'example.com'
>>> fqdn = zone + '.'
>>> gslb = GSLB(zone, fqdn)
```

Replacing a GSLB Monitor

If you'd like to create a brand new `Monitor` for your GSLB service, rather than update your existing one, the following example shows how simple it is to accomplish this task

```
>>> from dyn.tm.services.gslb import GSLB, Monitor
>>> zone = 'example.com'
>>> fqdn = zone + '.'
>>> gslb = GSLB(zone, fqdn)
>>> gslb.monitor.protocol
'HTTP'
>>> expected_text = "This is the text you're looking for."
>>> new_monitor = Monitor('HTTPS', 10, timeout=500, port=5005,
                          expected=expected_text)
>>> gslb.monitor = new_monitor
>>> gslb.monitor.protocol
'HTTPS'
```

Reverse DNS

class dyn.tm.services.reversedns.**ReverseDNS**(*zone*, *fqdn*, **args*, ***kwargs*)

A DynECT ReverseDNS service

__init__(*zone*, *fqdn*, **args*, ***kwargs*)

Create an new ReverseDNS object instance

Parameters

- **zone** – The zone under which this service will be attached
- **fqdn** – The fqdn where this service will be located
- **hosts** – A list of Hostnames of the zones where you want to track records
- **netmask** – A netmask to match A/AAAA rdata against. Matched records will get PTR records, any others won't
- **ttl** – TTL for the created PTR records. May be omitted, explicitly specified, set to 'default', or 'match'
- **record_types** – A list of which type of records this service will track. Note: Both A and AAAA can not be monitored by the same service

activate()

Activate this ReverseDNS service

active

Indicates whether or not the service is active. When setting directly, rather than using activate/deactivate valid arguments are 'Y' or True to activate, or 'N' or False to deactivate. Note: If your service is already active and you try to activate it, nothing will happen. And vice versa for deactivation.

Returns An Active object representing the current state of this ReverseDNS Service

deactivate()

Deactivate this ReverseDNS service

delete()

Delete this ReverseDNS service from the DynECT System

fqdn

The fqdn that this ReverseDNS Service is attached to is a read-only attribute

hosts

Hostnames of zones in your account where you want to track records

iptrack_id
The unique System id for this service. This is a read-only property.

netmask
A netmask to match A/AAAA rdata against. Matched records will get PTR records, any others won't

record_types
Types of records to track

ttl
TTL for the created PTR records. Omit to use zone default

zone
The zone that this ReverseDNS Service is attached to is a read-only attribute

Reverse DNS Examples

The following examples highlight how to use the `ReverseDNS` class to get/create `ReverseDNS`'s on the dyn.tm System and how to edit these objects from within a Python script.

Creating a new Reverse DNS Service

The following example shows how to create a new `ReverseDNS` on the dyn.tm System and how to edit some of the fields using the returned `ReverseDNS` object.

```
>>> from dyn.tm.services.reversedns import ReverseDNS
>>> # Create a dyn.tmSession
>>> # Assuming you own the zone 'example.com'
>>> rdns = ReverseDNS('example.com', 'example.com.', ['example.com],
...                   '127.0.0.0/8')
>>> rdns.deactivate()
>>> rdns.active
u'N'
```

Getting an Existing Reverse DNS Service

The following example shows how to get an existing `ReverseDNS` from the dyn.tm System and how to edit some of the same fields mentioned above.

```
>>> from dyn.tm.services.reversedns import ReverseDNS
>>> # Create a dyn.tmSession
>>> # Once again, assuming you own 'example.com'
>>> rdns = ReverseDNS('example.com', 'example.com.', my_rdns_id)
```

Real Time Traffic Management

The `services` module contains interfaces to all of the various service management features offered by the dyn.tm REST API

Monitor

class dyn.tm.services.rttm.**Monitor**(*protocol*, *interval*, *retries=None*, *timeout=None*, *port=None*, *path=None*, *host=None*, *header=None*, *expected=None*)

A Monitor for RTTM Service. May be used as a HealthMonitor

__init__(*protocol*, *interval*, *retries=None*, *timeout=None*, *port=None*, *path=None*, *host=None*, *header=None*, *expected=None*)

Create a Monitor object

Parameters

- **protocol** – The protocol to monitor. Must be either HTTP, HTTPS, PING, SMTP, or TCP
- **interval** – How often (in minutes) to run the monitor. Must be 1, 5, 10, or 15,
- **retries** – The number of retries the monitor attempts on failure before giving up
- **timeout** – The amount of time in seconds before the connection attempt times out
- **port** – For HTTP(S)/SMTP/TCP probes, an alternate connection port
- **path** – For HTTP(S) probes, a specific path to request
- **host** – For HTTP(S) probes, a value to pass in to the Host
- **header** – For HTTP(S) probes, additional header fields/values to pass in, separated by a newline character.
- **expected** – For HTTP(S) probes, a string to search for in the response. For SMTP probes, a string to compare the banner against. Failure to find this string means the monitor will report a down status.

expected

For HTTP(S) probes, a string to search for in the response. For SMTP probes, a string to compare the banner against. Failure to find this string means the monitor will report a down status

header

For HTTP(S) probes, additional header fields/values to pass in, separated by a newline character

host

For HTTP(S) probes, a value to pass in to the Host

interval

How often to run this monitor

path

For HTTP(S) probes, a specific path to request

port

For HTTP(S)/SMTP/TCP probes, an alternate connection port

protocol

The protocol to monitor

retries

The number of retries the monitor attempts on failure before giving up

status

Get the current status of this HealthMonitor from the DynECT System

timeout

The amount of time in seconds before the connection attempt times out

to_json()
Convert this HealthMonitor object to a JSON blob

RegionPoolEntry

class dyn.tm.services.rttm.**RegionPoolEntry**(*address*, *label*, *weight*, *serve_mode*, ***kwargs*)
Creates a new RTTM service region pool entry in the zone/node indicated

__init__(*address*, *label*, *weight*, *serve_mode*, ***kwargs*)
Create a RegionPoolEntry object

Parameters

- **address** – The IPv4 address or FQDN of this Node IP
- **label** – A descriptive string identifying this IP
- **weight** – A number from 1-15 describing how often this record should be served. The higher the number, the more often the address is served
- **serve_mode** – Sets the behavior of this particular record. Must be one of 'always', 'obey', 'remove', or 'no'

address
The IPv4 address or FQDN of this Node IP

delete()
Delete this RegionPoolEntry

fqdn
FQDN for this RegionPoolEntry, this is stored locally for REST command completion

label
A descriptive string identifying this IP

logs

region_code
region_code for this RegionPoolEntry, this is stored locally for REST command completion

serve_mode
Sets the behavior of this particular record

task
Task for most recent system action on this RegionPoolEntry.

to_json()
Return a JSON representation of this RegionPoolEntry

weight
A number from 1-15 describing how often this record should be served. The higher the number, the more often the address is served

zone
Zone for this RegionPoolEntry, this is stored locally for REST command completion

RTTMRegion

class dyn.tm.services.rttm.**RTTMRegion**(*zone*, *fqdn*, *region_code*, **args*, ***kwargs*)
docstring for RTTMRegion

__init__(*zone*, *fqdn*, *region_code*, **args*, ***kwargs*)
Create a `RTTMRegion` object

Parameters

- **region_code** – Name of the region
- **pool** – (*arg) The IP Pool list for this region
- **autopopulate** – If set to Y, this region will automatically be filled in from the global pool, and any other options passed in for this region will be ignored
- **ep** – Eligibility Pool - How many records will make it into the eligibility pool. The addresses that get chosen will be those that respond the fastest
- **apmc** – The minimum amount of IPs that must be in the up state, otherwise the region will be in failover
- **epmc** – The minimum amount of IPs that must be populated in the EP, otherwise the region will be in failover
- **serve_count** – How many records will be returned in each DNS response
- **failover_mode** – How the region should failover. Must be one of 'ip', 'cname', 'region', or 'global'
- **failover_data** – Dependent upon failover_mode. Must be one of ip', 'cname', 'region', or 'global'

apmc
The minimum amount of IPs that must be in the up state, otherwise the region will be in failover.

autopopulate
If set to Y, this region will automatically be filled in from the global pool, and any other options passed in for this region will be ignored. Must be either 'Y' or 'N'.

delete()
Delete an existing `RTTMRegion` object from the DynECT System

ep
Eligibility Pool - How many records will make it into the eligibility pool. The addresses that get chosen will be those that respond the fastest

epmc
The minimum amount of IPs that must be populated in the EP, otherwise the region will be in failover

failover_data
Dependent upon failover_mode. Must be one of ip', 'cname', 'region', or 'global'

failover_mode
How the region should failover. Must be one of 'ip', 'cname', 'region', or 'global'

pool
The IP Pool list for this `RTTMRegion`

serve_count
How many records will be returned in each DNS response

status
The current state of the region.

task
`Task` for most recent system action on this `ActiveFailover`.

Real Time Traffic Manager

class dyn.tm.services.rttm.**RTTM**(*zone*, *fqdn*, **args*, ***kwargs*)

__init__(*zone*, *fqdn*, **args*, ***kwargs*)
Create a RTTM object

Parameters

- **auto_recover** – Indicates whether or not the service should automatically come out of failover when the IP addresses resume active status or if the service should remain in failover until manually reset. Must be one of 'Y' or 'N'
- **ttl** – Time To Live in seconds of records in the service. Must be less than 1/2 of the Health Probe's monitoring interval. Must be one of 30, 60, 150, 300, or 450.
- **notify_events** – A list of the events which trigger notifications. Must be one of 'ip', 'svc', or 'nosrv'
- **syslog_server** – The Hostname or IP address of a server to receive syslog notifications on monitoring events
- **syslog_port** – The port where the remote syslog server listens for notifications
- **syslog_ident** – The ident to use when sending syslog notifications
- **syslog_facility** – The syslog facility to use when sending syslog notifications. Must be one of kern, user, mail, daemon, auth, syslog, lpr, news, uucp, cron, authpriv, ftp, ntp, security, console, local0, local1, local2, local3, local4, local5, local6, or local7
- **syslog_delivery** – The syslog delivery action type. 'all' will deliver notifications no matter what the endpoint state. 'change' (default) will deliver only on change in the detected endpoint state
- **region** – A list of RTTMRegion's
- **monitor** – The Monitor for this service
- **performance_monitor** – The performance monitor for the service
- **contact_nickname** – Name of contact to receive notifications
- **syslog_probe_fmt** – see below for format:
- **syslog_status_fmt** – see below for format:
- **syslog_rttm_fmt** – see below for format: Use the following format for syslog_xxxx_fmt paramaters. %hos hostname %tim current timestamp or monitored interval %reg region code %sta status %ser record serial %rda rdata %sit monitoring site %rti response time %msg message from monitoring %adr address of monitored node %med median value %rts response times (RTTM)
- **recovery_delay** – number of up status polling intervals to consider service up

activate()
Activate this RTTM Service

active
Returns whether or not this RTTM Service is currently active. When setting directly, rather than using activate/deactivate valid arguments are 'Y' or True to activate, or 'N' or False to deactivate. Note: If your service is already active and you try to activate it, nothing will happen. And vice versa for deactivation.

Returns An `Active` object representing the current state of this `ReverseDNS` Service

auto_recover
Indicates whether or not the service should automatically come out of failover when the IP addresses resume active status or if the service should remain in failover until manually reset. Must be one of 'Y' or 'N'

contact_nickname
The name of contact to receive notifications from this service

deactivate()
Deactivate this RTTM Service

delete()
Delete this RTTM Service

get_log_report(*start_ts*, *end_ts=None*)
Generates a report with information about changes to an existing RTTM service

Parameters

- **start_ts** – datetime.datetime instance identifying point in time for the start of the log report
- **end_ts** – datetime.datetime instance identifying point in time for the end of the log report. Defaults to datetime.datetime.now()

Returns dictionary containing log report data

get_rrset_report(*ts*)
Generates a report of regional response sets for this RTTM service at a given point in time

Parameters **ts** – UNIX timestamp identifying point in time for the log report

Returns dictionary containing rrset report data

monitor
The `Monitor` for this service

notify_events
A list of events which trigger notifications. Valid values are: 'ip', 'svc', and 'nosrv'

performance_monitor
The Performance `Monitor` for this service

recover(*recoverip=None*, *address=None*)
Recovers the RTTM service or a specific node IP within the service

recovery_delay

region
A list of `RTTMRegion`s

status
Status

syslog_delivery

syslog_facility
The syslog facility to use when sending syslog notifications. Must be one of kern, user, mail, daemon, auth, syslog, lpr, news, uucp, cron, authpriv, ftp, ntp, security, console, local0, local1, local2, local3, local4, local5, local6, or local7

syslog_ident
The ident to use when sending syslog notifications

syslog_port
The port where the remote syslog server listens for notifications

syslog_probe_format

syslog_rttm_format

syslog_server
The Hostname or IP address of a server to receive syslog notifications on monitoring events

syslog_status_format

task
`Task` for most recent system action on this `ActiveFailover`.

ttl
Time To Live in seconds of records in the service. Must be less than 1/2 of the Health Probe's monitoring interval. Must be one of 30, 60, 150, 300, or 450.

Real Time Traffic Manager Examples

The following examples highlight how to use the `RTTM` class to get/create `RTTM`'s on the dyn.tm System and how to edit these objects from within a Python script.

Creating a new Real Time Traffic Manager Service

The following example shows how to create a new `RTTM` on the dyn.tm System and how to edit some of the fields using the returned `RTTM` object.

```
>>> from dyn.tm.services.rttm import Monitor, RegionPoolEntry, RTTMRegion, \
...     RTTM
>>> # Create a dyn.tmSession
>>> # Assuming you own the zone 'example.com'
>>> zone = 'example.com'
>>> fqdn = zone + '.'
>>> entry = RegionPoolEntry('1.1.1.1', 'RPE Label', 5, 'always')
>>> region = RTTMRegion(zone, fqdn, 'global', [self.entry])
>>> monitor = Monitor('HTTP', 5, expected='Example')
>>> performance_monitor = Monitor('HTTP', 20)
>>> rttm = RTTM(zone, fqdn, 'mycontactname', region=[region],
...             monitor=monitor, performance_monitor=performance_monitor)
```

Getting an Existing Real Time Traffic Manager Service

The following example shows how to get an existing `RTTM` from the dyn.tm System and how to edit some of the same fields mentioned above.

```
>>> from dyn.tm.services.rttm import RTTM
>>> # Create a dyn.tmSession
>>> # Once again, assuming you own 'example.com'
>>> zone = 'example.com'
>>> fqdn = zone + '.'
>>> rttm = RTTM(zone, fqdn)
>>> rttm.notify_events
u'ip'
```

```
>>> rttm.notify_events = 'ip, nosrv'
>>> rttm.notify_events
u'ip, nosrv'
```

TM Reports

The `reports` module contains interfaces for all of the various Report collection calls offered by the dyn.tm REST API

List Functions

`dyn.tm.reports.`**`get_check_permission`**(*permission*, *zone_name=None*)
Returns a list of allowed and forbidden permissions for the currently logged in user based on the provided permissions array.

Parameters

- **`permission`** – A list of permissions to check for the current user.
- **`zone_name`** – The zone to check for specific permissions.

Returns A *dict* containing permission information.

`dyn.tm.reports.`**`get_dnssec_timeline`**(*zone_name*, *start_ts=None*, *end_ts=None*)
Generates a report of events for the *DNSSEC* service attached to the specified zone has performed and has scheduled to perform.

Parameters

- **`zone_name`** – The name of the zone with DNSSEC service
- **`start_ts`** – datetime.datetime instance identifying point in time for the start of the timeline report
- **`end_ts`** – datetime.datetime instance identifying point in time for the end of the timeline report. Defaults to datetime.datetime.now()

Returns A *dict* containing log report data

`dyn.tm.reports.`**`get_rttm_log`**(*zone_name*, *fqdn*, *start_ts*, *end_ts=None*)
Generates a report with information about changes to an existing RTTM service.

Parameters

- **`zone_name`** – The name of the zone
- **`fqdn`** – The FQDN where RTTM is attached
- **`start_ts`** – datetime.datetime instance identifying point in time for the log report to start
- **`end_ts`** – datetime.datetime instance indicating the end of the data range for the report. Defaults to datetime.datetime.now()

Returns A *dict* containing log report data

`dyn.tm.reports.`**`get_rttm_rrset`**(*zone_name*, *fqdn*, *ts*)
Generates a report of regional response sets for this RTTM service at a given point in time.

Parameters

- **zone_name** – The name of the zone
- **fqdn** – The FQDN where RTTM is attached
- **ts** – datetime.datetime instance identifying point in time for the report

Returns A *dict* containing rrset report data

dyn.tm.reports.**get_qps**(*start_ts*, *end_ts=None*, *breakdown=None*, *hosts=None*, *rrecs=None*, *zones=None*)

Generates a report with information about Queries Per Second (QPS).

Parameters

- **start_ts** – datetime.datetime instance identifying point in time for the QPS report
- **end_ts** – datetime.datetime instance indicating the end of the data range for the report. Defaults to datetime.datetime.now()
- **breakdown** – By default, most data is aggregated together. Valid values ('hosts', 'rrecs', 'zones').
- **hosts** – List of hosts to include in the report.
- **rrecs** – List of record types to include in report.
- **zones** – List of zones to include in report.

Returns A *str* with CSV data

dyn.tm.reports.**get_zone_notes**(*zone_name*, *offset=None*, *limit=None*)

Generates a report containing the Zone Notes for given zone.

Parameters

- **zone_name** – The name of the zone
- **offset** – UNIX timestamp of the starting point at which to retrieve the notes
- **limit** – The maximum number of notes to be retrieved

Returns A *list* of *dict* containing Zone Notes

TM Tools

The tools module contains utility functions for performing common and potentially difficult tasks easily.

List Functions

dyn.tm.tools.**change_ip**(*zone*, *from_ip*, *to*, *v6=False*, *publish=False*)

Change all occurances of an ip address to a new ip address under the specified zone

Parameters

- **zone** – The *Zone* you wish to update ips for
- **from_ip** – Either a list of ip addresses or a single ip address that you want updated
- **to** – Either a list of ip addresses or a single ip address that will overwrite from_ip
- **v6** – Boolean flag to specify if we're replacing ipv4 or ipv6 addresses (ie, whether we're updating an ARecord or AAAARecord)

- **publish** – A boolean flag denoting whether or not to publish changes after making them. You can optionally leave this as *False* and process the returned changeset prior to publishing your changes.

Returns A list of tuples of the form (fqdn, old, new) where fqdn is the fqdn of the record that was updated, old was the old ip address, and new is the new ip address.

dyn.tm.tools.**map_ips**(*zone*, *mapping*, *v6=False*, *publish=False*)
Change all occurances of an ip address to a new ip address under the specified zone

Parameters

- **zone** – The *Zone* you wish to update ips for
- **mapping** – A *dict* of the form {'old_ip': 'new_ip'}
- **v6** – Boolean flag to specify if we're replacing ipv4 or ipv6 addresses (ie, whether we're updating an ARecord or AAAARecord)
- **publish** – A boolean flag denoting whether or not to publish changes after making them. You can optionally leave this as *False* and process the returned changeset prior to publishing your changes.

Returns A list of tuples of the form (fqdn, old, new) where fqdn is the fqdn of the record that was updated, old was the old ip address, and new is the new ip address.

Tools Examples

change_ip

If you find yourself replacing a server with a new one, or in some other situation where you might want to replace an ip address with a new one, then `change_ip()` makes it straight forward to apply these changes

```
>>> from dyn.tm.zones import Zone
>>> from dyn.tm.tools import change_ip
>>> my_zone = Zone('example.com')
>>> old = '1.1.1.1'
>>> new = '1.1.1.2'
>>> change_ip(my_zone, old, new, publish=True)
```

This handles acquiring and ARecords under the provided zone and applying the changes as you've specified. Need to shift over a handful of ip addresses?

```
>>> from dyn.tm.zones import Zone
>>> from dyn.tm.tools import change_ip
>>> my_zone = Zone('example.com')
>>> old = ['1.1.1.1', '1.1.1.3', '1.1.1.5']
>>> new = ['1.1.1.2', '1.1.1.4', '1.1.1.6']
>>> change_ip(my_zone, old, new, publish=True)
```

Have IPv6 addresses you need to switch over?

```
>>> from dyn.tm.zones import Zone
>>> from dyn.tm.tools import change_ip
>>> my_zone = Zone('example.com')
>>> old = '::1'
>>> new = '2001:db8:85a3::8a2e:370:7334'
>>> change_ip(my_zone, old, new, v6=True, publish=True)
```

Don't want to automatically publish, but rather wait and validate the changes manually?

```
>>> from dyn.tm.zones import Zone
>>> from dyn.tm.tools import change_ip
>>> my_zone = Zone('example.com')
>>> old = '1.1.1.1'
>>> new = '1.1.1.2'
>>> changeset = change_ip(my_zone, old, new)
>>> changeset
[(u'example.com.', u'1.1.1.1', u'1.1.1.2')]
```

map_ips

`map_ips()` functions in basically the same manner as `change_ip()`, the only difference being that it accepts a *dict* with rules on mapping form one ip to another (as well as the same v6 flag for specifying that you're working ipv6 addresses.

```
>>> from dyn.tm.zones import Zone
>>> from dyn.tm.tools import map_ips
>>> my_zone = Zone('example.com')
>>> old = '1.1.1.1'
>>> new = '1.1.1.2'
>>> mapping = {old: new}
>>> map_ips(my_zone, mapping, publish=True)
```

TM Errors

dyn.tm.errors module

This module contains all DynectDNS Errors. Each Error subclass inherits from the base DynectError class which is only ever directly raised if something completely unexpected happens TODO: add a DynectInvalidPermissionsError

exception dyn.tm.errors.**DynectAuthError**(**args*, ***kwargs*)
Error raised if Authentication to Dynect failed

__init__(**args*, ***kwargs*)
Format this errors message to report back the JSON messages returned from a faulty Session POST

exception dyn.tm.errors.**DynectInvalidArgumentError**(*arg*, *value*, *valid_args=None*)
Error raised if a given argument is determined to be invalid

__init__(*arg*, *value*, *valid_args=None*)
Format this error's message to report back the invalid argument and a list of valid arguments, if such a list exists

exception dyn.tm.errors.**DynectCreateError**(**args*, ***kwargs*)
Error raised if an API POST method returns with a failure

__init__(**args*, ***kwargs*)
Format this error's message to report back the JSON error message(s)

exception dyn.tm.errors.**DynectUpdateError**(**args*, ***kwargs*)
Error raised if an API PUT method returns with a failure

__init__(**args*, ***kwargs*)
Format this error's message to report back the JSON error message(s)

exception dyn.tm.errors.**DynectGetError**(**args*, ***kwargs*)
Error raised if an API PUT method returns with a failure

__init__(**args*, ***kwargs*)
Format this error's message to report back the JSON error message(s)

exception dyn.tm.errors.**DynectDeleteError**(**args*, ***kwargs*)
Error raised if an API DELETE method returns with a failure

__init__(**args*, ***kwargs*)
Format this error's message to report back the JSON error message(s)

exception dyn.tm.errors.**DynectQueryTimeout**(**args*, ***kwargs*)
Error raised if an API call times out even after waiting for a response

__init__(**args*, ***kwargs*)
Format this error's message to report back the JSON error message(s)

CHAPTER 5

dyn.mm (Message Management) Module

The dyn.mm (MM) module provides access to all of the Message Management resources provided by Dyn's Message Management REST API. It's important to note that all code examples assume the existence of a *MMSession* instance. This object is used by the modules described below to access the API and make their associated calls. If you are looking for information on a specific function, class or method, this part of the documentation is for you.

MM Accounts

The `accounts` module contains interfaces for all of the various Account management features offered by the dyn Message Management REST API

Search/List Functions

The following functions return a single `list` containing class representations of their respective types. For instance `get_all_users()` returns a `list` of `User` objects.

Account

Create a new Account

The following example shows how to create a new `Account` on the Dyn Message Management system:

```
>>> from dyn.mm.accounts import Account
>>> new_account = Account('username', 'password', 'companyname', '1 (603) 867-5309')
>>> new_account
<MM Account>: username
>>> new_account.xheaders
{}
```

Using an Existing Account

The following example shows how to get an do some simple manipulation of an existing dyn Message Management account:

```
>>> from dyn.mm.accounts import Account
>>> new_account = Account('username')
>>> new_account.apikey
'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX'
>>> new_account.generate_new_apikey()
>>> new_account.apikey
'YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY'
>>> new_account.xheaders
{'xheader1': '', 'xheader2': '', 'xheader3': '', 'xheader4': ''}
>>> # The following creates a new xheader for the account
>>> new_account.xheaders['xheader3'] = 'X-header3_data'
```

Approved Sender

Create a new Approved Sender

Approved senders are pretty straightforward as far as functionality goes but here we'll see how to create a new `ApprovedSender`:

```
>>> from dyn.mm.accounts import ApprovedSender
>>> sender = ApprovedSender('username@email.com', seeding=0)
>>> sender.status
1
>>> sender.seeding
0
>>> sender.seeding = 1
```

Recipient

Creating/Using Recipients

Recipients are the one model you'll find in this library that don't have an intuitive way to distinguish what you're trying to accomplish simply from the arguments you provide at create time. Because of this, you'll need to pass a method type, either GET or POST, to the Recipient when you create it:

```
>>> from dyn.mm.accounts import Recipient
>>> recipient = Recipient('user@email.com', method='POST')
>>> recipient.status
'inactive'
>>> recipient.activate()
>>> recipient.status
'active'
```

Suppression

Messages

The dyn.mm.message module is where you'll find the ability to easily automate the sending of messages.

send_message

The *send_message()* function allows a user to quickly fire off an email via the Message Management API

dyn.mm.message.**send_message**(*from_field*, *to*, *subject*, *cc=None*, *body=None*, *html=None*, *replyto=None*, *xheaders=None*)

Create and send an email on the fly. For information on the arguments accepted by this function see the documentation for *EMail*

Using send_message

Below is just a quick example on how to use *send_message()*:

```
>>> from dyn.mm.message import send_message
>>> from_email = 'user@email.com'
>>> to_email = 'your@email.com'
>>> subject = 'A Demo Email'
>>> content = 'Hello User, thank you for registering at http://mysite.com!'
>>> send_message(from_email, to_email, subject, body=content)
```

EMail

class dyn.mm.message.**EMail**(*from_field*, *to*, *subject*, *cc=None*, *body=None*, *html=None*, *replyto=None*, *xheaders=None*)

Create an and Send it from one of your approved senders

__init__(*from_field*, *to*, *subject*, *cc=None*, *body=None*, *html=None*, *replyto=None*, *xheaders=None*)

Create a new *EMail* object

Parameters

- **from_field** – Sender email address - This can either be an email address or a properly formatted from header (example: "From Name" <example@email.com>). NOTE: The sender must be one of your account's Approved Senders
- **to** – A *list* of Address(es) or a single Address that the email will be sent to — This/These can either be an email address or a properly formatted from header (example: "To Name" <example@email.com>). The To field in the email will contain all the addresses when it is sent out and will be sent to all the addresses.
- **subject** – The subject of the email being sent
- **cc** – Address(es) to copy the email to - This can either be an email address or a properly formatted cc header (example: "cc Name" <example@email.com>). For multiple addresses, each address must have its own 'cc' field. (example: cc = "example1@email.com", cc = "example2@email.com").
- **body** – The plain/text version of the email; this field may be encoded in Base64 (recommended), quoted-printable, 8-bit, or 7-bit.

- **html** – The text/html version of the email; this field may be encoded in 7-bit, 8-bit, quoted-printable, or base64.
- **replyto** – The email address for the recipient to reply to. If left blank, defaults to the from address.
- **xheaders** – Any additional custom X-headers to send in the email - Pass the X-header's name as the field name and the X-header's value as the value (example: x-demonheader=zoom).

send(*content=None*)
Send the content of this Email object to the provided list of recipients.

Parameters **content** – The optional content field can be used to overrwrite, or to specify the actual content of the body of the message. Note: If *content*, this instance's body, and this instance's html fields are all *None*, then a *DynInvalidArgumentError* will be raised.

uri = '/send'

Using the EMail Base class

The ability to be able to customize your messages become far more apparent with the use of the *EMail* class as you can see in the example below it's very easy to use this class for templating, or even subclassing to make sending emails quick and easy:

```
>>> from dyn.mm.message import EMail
>>> from_email = 'user@email.com'
>>> to_email = 'your@email.com'
>>> subject = 'A Demo Email'
>>> content = 'Hello %s, thank you for registering at http://mysite.com!'
>>> mailer = EMail(from_email, to_email, subject)
>>> user_names = ['Jon', 'Ray', 'Carol', 'Margaret']
>>> for user_name in user_names:
...     mailer.body = content % user_name
...     mailer.send()
```

EMail Subclasses

Below are some *EMail* subclasses which provide some additional formatting and, hopefully, helpful features.

class dyn.mm.message.**HTMLEMail**(*from_field*, *to*, *subject*, *cc=None*, *body=None*, *html=None*, *replyto=None*, *xheaders=None*)
EMail subclass with an overridden send method for specifying html content on the fly

send(*content=None*)
Send the content of this Email object to the provided list of recipients.

Parameters **content** – The optional content field can be used to overrwrite, or to specify the actual content of the html of the message. Note: If *content*, this instance's body, and this instance's html fields are all *None*, then a *DynInvalidArgumentError* will be raised.

class dyn.mm.message.**TemplateEMail**(*from_field*, *to*, *subject*, *cc=None*, *body=None*, *html=None*, *replyto=None*, *xheaders=None*)
EMail subclass which treats it's bodytext attribute as a template. Allowing you to send out chains of emails by only writing the templated email once, and then specifying an iterable with the formatting content at send time.

send(*formatters=None*)
Send the content of this `Email` object to the provided list of recipients.

Parameters **formatters** – Any iterable containing the data you wish inserted into your template. Unlike in the *EMail* class this field is not optional and will raise a *DynInvalidArgumentError* if not provided. This exception will also be raised if this instances bodytext attribute has not also been set.

class dyn.mm.message.**HTMLTemplateEMail**(*from_field*, *to*, *subject*, *cc=None*, *body=None*, *html=None*, *replyto=None*, *xheaders=None*)
Similar to the *TemplateEMail* class the *HTMLEMail* subclass which treats it's bodyhtml attribute as a template. Allowing you to send out chains of emails by only writing the templated html email once, and then specifying an iterable with the formatting content at send time.

send(*formatters=None*)
Send the content of this `Email` object to the provided list of recipients.

Parameters **formatters** – Any iterable containing the data you wish inserted into your html template. Unlike in the *HTMLEMail* class this field is not optional and will raise a *DynInvalidArgumentError* if not provided. This exception will also be raised if this instances bodyhtml attribute has not also been set.

Using the EMail Subclasses

The *HTMLEMail* class is identical to the *EMail* class, with the only difference being that content passed to it's send method will be added as the messages HTML content, rather than text content.

The Templating subclasses behave slightly differently. For the `TemplateEmail` class, you provide it a template at construction time, and an iterable with the content to substitute into the template at send time. For example:

```
>>> from dyn.mm.message import TemplateEmail
>>> from_email = 'user@email.com'
>>> to_email = 'your@email.com'
>>> subject = 'A Demo Email'
>>> template = 'Hello %s, thank you for registering at http://mysite.com!'
>>> mailer = TemplateEmail(from_email, to_email, subject, body=template)
>>> parameters = ['Jon', 'Ray', 'Carol', 'Margaret']
>>> mailer.send(parameters)
```

Similarly you can use the *HTMLTemplateEMail* class to template out and send multiple HTML formatted emails easily. Let's go over a slightly more complex for that class:

```
>>> from textwrap import dedent
>>> from dyn.mm.message import TemplateEmail
>>> from_email = 'user@email.com'
>>> to_email = 'your@email.com'
>>> subject = 'A Demo Email'
>>> template = """
<html>
    <h1>What... is the air-speed velocity of an unladen swallow?</h1>
    <h2>What do you mean? An %(choice1) or %(choice2) swallow?</h2>
</html>"""
>>> template = dedent(template)
>>> mailer = HTMLTemplateEmail(from_email, to_email, subject, html=template)
>>> parameters = {'choice1': 'African', 'choice2': 'European'}
>>> mailer.send(parameters)
```

Reports

dyn.mm.reports module

MM Session

dyn.mm.session module

This module implements an interface to a DynECT REST Session. It provides easy access to all other functionality within the dynect library via methods that return various types of DynECT objects which will provide their own respective functionality.

class dyn.mm.session.**MMSession**(*apikey*, *host='emailapi.dynect.net'*, *port=443*, *ssl=True*, *proxy_host=None*, *proxy_port=None*, *proxy_user=None*, *proxy_pass=None*)

Base object representing a Message Management API Session

__init__(*apikey*, *host='emailapi.dynect.net'*, *port=443*, *ssl=True*, *proxy_host=None*, *proxy_port=None*, *proxy_user=None*, *proxy_pass=None*)

Initialize a Dynect Rest Session object and store the provided credentials

Parameters

- **host** – DynECT API server address
- **port** – Port to connect to DynECT API server
- **ssl** – Enable SSL
- **apikey** – your unique Email API key
- **proxy_host** – A proxy host to utilize
- **proxy_port** – The port that the proxy is served on
- **proxy_user** – A username to connect to the proxy with if required
- **proxy_pass** – A password to connect to the proxy with if required

uri_root = '/rest/json'

MM Errors

Below are the various errors you may see be raised while using the dyn.mm module along with brief descriptions about when those exceptions are raised.

dyn.mm.errors module

This module contains all Dyn Email Errors. Each Error subclass inherits from the base EmailError class which is only ever directly raised if something completely unexpected happens

exception dyn.mm.errors.**EmailKeyError**(*reason*)

Error raised if the associated API Key is missing or invalid

exception dyn.mm.errors.**DynInvalidArgumentError**(*arg*, *value*, *valid_args=None*)

Error raised if a given argument is determined to be invalid

__init__(*arg*, *value*, *valid_args=None*)
Format this error's message to report back the invalid argument and a list of valid arguments, if such a list exists

exception dyn.mm.errors.**EmailInvalidArgumentError**(*reason*)
Error raised if a required field is not provided. However, due to the nature or the wrapper being used this error is most likely caused but uncaught invalid input (i.e., letters instead of numbers, etc.).

exception dyn.mm.errors.**EmailObjectError**(*reason*)
This error can come up if you try to create an object that already exists on the Dyn Email system.

exception dyn.mm.errors.**NoSuchAccountError**(*reason*)
Error raised if you attempt to GET an Account that does not exist, or is not accessible to you

CHAPTER 6

Advanced Topics

This section is a collection of advanced topics for users who intend to contribute and maintain this library.

Sessions

Sessions in this library are designed for ease of use by front-end users. However, this section is dedicated to a deeper understanding of Sessions for advanced users and contributors to this library.

Parent Class

Both *dyn.tm.session.DynectSession* and *dyn.mm.session.MMSession* are subclasses of *dyn.core.SessionEngine*. The *dyn.core.SessionEngine* provides a simple internal API for preparing, sending, and processing outbound API calls. This class was added in v1.0.0 and reduced the amount of logic and duplicated code that made understanding these Sessions difficult.

Parent Type

Since v0.4.0, Sessions have been implemented as a Singleton type. This made it easier for end users to use the SDK and to utilize the API. By internally implementing Sessions as a Singleton, it allows the user discard their Session objects, unless they wish to keep them. It also doesn't require users to share their Session information with other classes in this library to make API calls. (EXAMPLE):

```
>>> from dyn.tm.session import DynectSession
>>> from dyn.tm.zones import get_all_zones
>>> DynectSession(**my_credentials)
>>> zones = get_all_zones()
```

as opposed to something like this:

```
>>> from dyn.tm.session import DynectSession
>>> from dyn.tm.zones import get_all_zones
>>> my_session = DynectSession(**my_credentials)
>>> zones = get_all_zones(my_session)
```

Or, even worse:

```
>>> from dyn.tm.session import DynectSession
>>> my_session = DynectSession(**my_credentials)
>>> zones = my_session.get_all_zones(my_session)
```

In these examples, the changes may not seem significant but gain more relevance when creating multiple types of records, adding or editing Traffic Director and other complex services. Not needing to share your Session with other classes, or use it as a point of entry to other functionality, makes using this SDK much simpler.

What We Used to Do

From a backend perspective, the following is an example of how Session types were handled before v0.4.0:

```
def session():
    """Accessor for the current Singleton DynectSession"""
    try:
        return globals()['SESSION']
    except KeyError:
        return None


class DynectSession(object):
    """Base object representing a DynectSession Session"""
    def __init__(self, customer, username, password, host='api.dynect.net',
                 port=443, ssl=True, api_version='current', auto_auth=True):
        # __init__ logic here

    def __new__(cls, *args, **kwargs):
        try:
            if globals()['SESSION'] is None:
                globals()['SESSION'] = super(DynectSession, cls).__new__(cls,
                                                                         *args,
                                                                         **kwargs)
        except KeyError:
            globals()['SESSION'] = super(DynectSession, cls).__new__(cls, *args)
        return globals()['SESSION']
```

While this worked for a short while, it had its flaws:

1. Once Message Management support was added, the code needed to be duplicated to rename the 'SESSION' key to 'MM_SESSION'. This was inefficient.
2. This allowed you to only have one active Session, even in shared memory space, i.e. threads.
3. Sessions were only truly "global" in the scope of the dyn.tm module. It could still be accessed externally, but it was less than ideal.

What We Do Now

As of v1.0.0, Session types remain Singletons but are implemented differently.

Sessions are now implemented as *dyn.core.SessionEngine* objects and *dyn.core.Singleton* *type* objects. EXAMPLE:

```
class Singleton(type):
    """A :class:`Singleton` type for implementing a true Singleton design
    pattern, cleanly, using metaclasses
    """
    _instances = {}
    def __call__(cls, *args, **kwargs):
        cur_thread = threading.current_thread()
        key = getattr(cls, '__metakey__')
        if key not in cls._instances:
            cls._instances[key] = {
                # super(Singleton, cls) evaluates to type; *args/**kwargs get
                # passed to class __init__ method via type.__call__
                cur_thread: super(_Singleton, cls).__call__(*args, **kwargs)
            }
        return cls._instances[key][cur_thread]
```

The Singleton type is applied as a *__metaclass__* in each of the two Session types. This allows for a much cleaner implementation of Singletons. Every time one is accessed, it will globally have knowledge of other instances, as those instances are tied to the classes themselves instead of held in the *globals* of the session modules. In addition, this allows users to have multiple active sessions across multiple threads, which was not possible in the prior implementation.

Password Encryption

The Managed DNS REST API only accepts passwords in plain text. The passwords stored in *DynectSession* objects only live in memory, reducing the security risk of plain text passwords in this instance. However, for users looking to do more advanced things, such as serialize and store their session objects in something less secure, such as a database, these plain text passwords are not ideal. In response to this, Dyn added optional AES-256 password encryption for all *DynectSession* instances in version 1.1.0. To enable password encryption, install PyCrypto.

Key Generation

In version 1.1.0, an optional key field parameter was added to the *DynectSession* __init__ method. This field will allow you to specify the key that your encrypted password will be using. You can also let the Dyn module handle the key generation in addition to using the `generate_key()` function, which generates a random 50 character key that can be easily consumed by the *AESCipher* class (the class responsible for performing the encryption and decryption).

Encrypt Module

:: .. autofunction:: dyn.encrypt.generate_key

class dyn.encrypt.**AESCipher**(*key=None*)
An AES-256 password hasher

__init__(*key=None*)
Create a new AES-256 Cipher instance

Parameters **key** – The secret key used to generate the password hashes

decrypt(*enc*)
Decrypt an encoded password hash using the secret key provided, and return the decrypted string

Parameters **enc** – The encoded AES-256 password hash

encrypt(*raw*)

Encrypt the provided password and return the encoded password hash

Parameters **raw** – The raw password string to encode

CHAPTER 7

Core

The `dyn.core` module contains functionality that is core to the behavior of the rest of the library. This is where a lot of the "heavy lifting" for sessions is done.

Singleton

class `dyn.core.`**`Singleton`**
A *Singleton* type for implementing a true Singleton design pattern, cleanly, using metaclasses

SessionEngine

class `dyn.core.`**`SessionEngine`**(*host=None*, *port=443*, *ssl=True*, *history=False*, *proxy_host=None*, *proxy_port=None*, *proxy_user=None*, *proxy_pass=None*)
Base object representing a DynectSession Session

`__init__`(*host=None*, *port=443*, *ssl=True*, *history=False*, *proxy_host=None*, *proxy_port=None*, *proxy_user=None*, *proxy_pass=None*)
Initialize a Dynect Rest Session object and store the provided credentials

Parameters

- **`host`** – DynECT API server address
- **`port`** – Port to connect to DynECT API server
- **`ssl`** – Enable SSL
- **`history`** – A boolean flag determining whether or not you would like to store a record of all API calls made to review later
- **`proxy_host`** – A proxy host to utilize
- **`proxy_port`** – The port that the proxy is served on
- **`proxy_user`** – A username to connect to the proxy with if required

- **proxy_pass** – A password to connect to the proxy with if required

Returns SessionEngine object

classmethod close_session()
Remove the current session from the dict of instances and return it. If there was not currently a session being stored, return None. If, after removing this session, there is nothing under the current key, delete that key's entry in the _instances dict.

connect()
Establishes a connection to the REST API server as defined by the host, port and ssl instance variables. If a proxy is specified, it is used.

execute(*uri*, *method*, *args=None*, *final=False*)
Execute a commands against the rest server

Parameters

- **uri** – The uri of the resource to access. /REST/ will be prepended if it is not at the beginning of the uri
- **method** – One of 'DELETE', 'GET', 'POST', or 'PUT'
- **args** – Any arguments to be sent as a part of the request
- **final** – boolean flag representing whether or not we have already failed executing once or not

classmethod get_session()
Return the current session for this Session type or None if there is not an active session

history
A history of all API calls that have been made during the duration of this Session's existence. These API call details are returned as a *list* of 5-tuples of the form: (timestamp, uri, method, args, status) where status will be one of 'success' or 'failure'

name
A human readable version of the name of this object

classmethod new_session(**args*, ***kwargs*)
Return a new session instance, regardless of whether or not there is already an existing session.

Parameters

- **args** – Arguments to be passed to the Singleton __call__ method
- **kwargs** – keyword arguments to be passed to the Singleton __call__ method

poll_response(*response*, *body*)
Looks at a response from a REST command, and while indicates that the job is incomplete, poll for response

Parameters

- **response** – the JSON response containing return codes
- **body** – the body of the HTTP response

send_command(*uri*, *method*, *args*)
Responsible for packaging up the API request and sending it to the server over the established connection

Parameters

- **uri** – The uri of the resource to interact with
- **method** – The HTTP method to use

- **args** – Encoded arguments to send to the server

uri_root = '/'

wait_for_job_to_complete(*job_id*, *timeout=120*)

When a response comes back with a status of "incomplete" we need to wait and poll for the status of that job until it comes back with success or failure

Parameters

- **job_id** – the id of the job to poll for a response from
- **timeout** – how long (in seconds) we should wait for a valid response before giving up on this request

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

dyn.mm.errors, 124
dyn.mm.session, 124
dyn.tm.errors, 117

Index

Symbols

__init__() (dyn.core.SessionEngine method), 131
__init__() (dyn.encrypt.AESCipher method), 129
__init__() (dyn.mm.errors.DynInvalidArgumentError method), 124
__init__() (dyn.mm.message.EMail method), 121
__init__() (dyn.mm.session.MMSession method), 124
__init__() (dyn.tm.accounts.Contact method), 33
__init__() (dyn.tm.accounts.Notifier method), 32
__init__() (dyn.tm.accounts.PermissionsGroup method), 29
__init__() (dyn.tm.accounts.UpdateUser method), 24
__init__() (dyn.tm.accounts.User method), 25
__init__() (dyn.tm.errors.DynectAuthError method), 117
__init__() (dyn.tm.errors.DynectCreateError method), 117
__init__() (dyn.tm.errors.DynectDeleteError method), 118
__init__() (dyn.tm.errors.DynectGetError method), 118
__init__() (dyn.tm.errors.DynectInvalidArgumentError method), 117
__init__() (dyn.tm.errors.DynectQueryTimeout method), 118
__init__() (dyn.tm.errors.DynectUpdateError method), 117
__init__() (dyn.tm.records.AAAARecord method), 36
__init__() (dyn.tm.records.ALIASRecord method), 37
__init__() (dyn.tm.records.ARecord method), 36
__init__() (dyn.tm.records.CAARecord method), 37
__init__() (dyn.tm.records.CDNSKEYRecord method), 39
__init__() (dyn.tm.records.CDSRecord method), 40
__init__() (dyn.tm.records.CERTRecord method), 38
__init__() (dyn.tm.records.CNAMERecord method), 41
__init__() (dyn.tm.records.CSYNCRecord method), 39
__init__() (dyn.tm.records.DHCIDRecord method), 41
__init__() (dyn.tm.records.DNAMERecord method), 41
__init__() (dyn.tm.records.DNSKEYRecord method), 42
__init__() (dyn.tm.records.DNSRecord method), 35
__init__() (dyn.tm.records.DSRecord method), 43
__init__() (dyn.tm.records.IPSECKEYRecord method), 45
__init__() (dyn.tm.records.KEYRecord method), 43
__init__() (dyn.tm.records.KXRecord method), 44
__init__() (dyn.tm.records.LOCRecord method), 45
__init__() (dyn.tm.records.MXRecord method), 46
__init__() (dyn.tm.records.NAPTRRecord method), 47
__init__() (dyn.tm.records.NSAPRecord method), 49
__init__() (dyn.tm.records.NSRecord method), 50
__init__() (dyn.tm.records.PTRRecord method), 48
__init__() (dyn.tm.records.PXRecord method), 48
__init__() (dyn.tm.records.RPRecord method), 49
__init__() (dyn.tm.records.SOARecord method), 50
__init__() (dyn.tm.records.SPFRecord method), 51
__init__() (dyn.tm.records.SRVRecord method), 51
__init__() (dyn.tm.records.TLSARecord method), 52
__init__() (dyn.tm.records.TXTRecord method), 52
__init__() (dyn.tm.services.active_failover.ActiveFailover method), 55
__init__() (dyn.tm.services.active_failover.HealthMonitor method), 54
__init__() (dyn.tm.services.ddns.DynamicDNS method), 58
__init__() (dyn.tm.services.dnssec.DNSSEC method), 60
__init__() (dyn.tm.services.dnssec.DNSSECKey method), 60
__init__() (dyn.tm.services.dsf.DSFAAAARecord method), 64
__init__() (dyn.tm.services.dsf.DSFALIASRecord method), 64
__init__() (dyn.tm.services.dsf.DSFARecord method), 63
__init__() (dyn.tm.services.dsf.DSFCERTRecord method), 65
__init__() (dyn.tm.services.dsf.DSFCNAMERecord method), 65
__init__() (dyn.tm.services.dsf.DSFDHCIDRecord method), 66
__init__() (dyn.tm.services.dsf.DSFDNAMERecord method), 66

__init__() (dyn.tm.services.dsf.DSFDNSKEYRecord method), 66
__init__() (dyn.tm.services.dsf.DSFDSRecord method), 67
__init__() (dyn.tm.services.dsf.DSFFailoverChain method), 80
__init__() (dyn.tm.services.dsf.DSFIPSECKEYRecord method), 70
__init__() (dyn.tm.services.dsf.DSFKEYRecord method), 68
__init__() (dyn.tm.services.dsf.DSFKXRecord method), 68
__init__() (dyn.tm.services.dsf.DSFLOCRecord method), 69
__init__() (dyn.tm.services.dsf.DSFMXRecord method), 70
__init__() (dyn.tm.services.dsf.DSFMonitor method), 90
__init__() (dyn.tm.services.dsf.DSFNAPTRRecord method), 71
__init__() (dyn.tm.services.dsf.DSFNSAPRecord method), 72
__init__() (dyn.tm.services.dsf.DSFNSRecord method), 73
__init__() (dyn.tm.services.dsf.DSFNotifier method), 91
__init__() (dyn.tm.services.dsf.DSFPTRRecord method), 71
__init__() (dyn.tm.services.dsf.DSFPXRecord method), 72
__init__() (dyn.tm.services.dsf.DSFRPRecord method), 73
__init__() (dyn.tm.services.dsf.DSFRecordSet method), 77
__init__() (dyn.tm.services.dsf.DSFResponsePool method), 83
__init__() (dyn.tm.services.dsf.DSFRuleset method), 86
__init__() (dyn.tm.services.dsf.DSFSPFRecord method), 74
__init__() (dyn.tm.services.dsf.DSFSRVRecord method), 74
__init__() (dyn.tm.services.dsf.DSFSSHFPRecord method), 75
__init__() (dyn.tm.services.dsf.DSFTXTRecord method), 75
__init__() (dyn.tm.services.dsf.TrafficDirector method), 93
__init__() (dyn.tm.services.gslb.GSLB method), 103
__init__() (dyn.tm.services.gslb.GSLBRegion method), 102
__init__() (dyn.tm.services.gslb.GSLBRegionPoolEntry method), 101
__init__() (dyn.tm.services.gslb.Monitor method), 100
__init__() (dyn.tm.services.reversedns.ReverseDNS method), 106
__init__() (dyn.tm.services.rttm.Monitor method), 108
__init__() (dyn.tm.services.rttm.RTTM method), 111
__init__() (dyn.tm.services.rttm.RTTMRegion method), 109
__init__() (dyn.tm.services.rttm.RegionPoolEntry method), 109
__init__() (dyn.tm.session.DynectSession method), 11
__init__() (dyn.tm.zones.Node method), 20
__init__() (dyn.tm.zones.SecondaryZone method), 19
__init__() (dyn.tm.zones.Zone method), 14

A

AAAARecord (class in dyn.tm.records), 36
activate() (dyn.tm.services.active_failover.ActiveFailover method), 56
activate() (dyn.tm.services.ddns.DynamicDNS method), 58
activate() (dyn.tm.services.dnssec.DNSSEC method), 60
activate() (dyn.tm.services.gslb.GSLB method), 104
activate() (dyn.tm.services.reversedns.ReverseDNS method), 106
activate() (dyn.tm.services.rttm.RTTM method), 111
activate() (dyn.tm.zones.SecondaryZone method), 19
active (dyn.tm.services.active_failover.ActiveFailover attribute), 56
active (dyn.tm.services.ddns.DynamicDNS attribute), 58
active (dyn.tm.services.dnssec.DNSSEC attribute), 60
active (dyn.tm.services.dsf.DSFMonitor attribute), 90
active (dyn.tm.services.gslb.GSLB attribute), 104
active (dyn.tm.services.reversedns.ReverseDNS attribute), 106
active (dyn.tm.services.rttm.RTTM attribute), 111
active (dyn.tm.zones.SecondaryZone attribute), 19
ActiveFailover (class in dyn.tm.services.active_failover), 55
add_failover_ip() (dyn.tm.services.dsf.DSFRuleset method), 86
add_forbid_rule() (dyn.tm.accounts.User method), 26
add_node() (dyn.tm.services.dsf.TrafficDirector method), 93
add_notifier() (dyn.tm.services.dsf.TrafficDirector method), 93
add_permission() (dyn.tm.accounts.PermissionsGroup method), 30
add_permission() (dyn.tm.accounts.User method), 26
add_permissions_group() (dyn.tm.accounts.User method), 26
add_recipient() (dyn.tm.services.dsf.DSFNotifier method), 92
add_record() (dyn.tm.zones.Node method), 21
add_record() (dyn.tm.zones.Zone method), 15
add_response_pool() (dyn.tm.services.dsf.DSFRuleset method), 86
add_service() (dyn.tm.zones.Node method), 21
add_service() (dyn.tm.zones.Zone method), 15

add_subgroup() (dyn.tm.accounts.PermissionsGroup method), 30
add_to_failover_chain() (dyn.tm.services.dsf.DSFRecordSet method), 77
add_to_response_pool() (dyn.tm.services.dsf.DSFFailoverChain method), 80
add_zone() (dyn.tm.accounts.PermissionsGroup method), 30
add_zone() (dyn.tm.accounts.User method), 26
address (dyn.tm.accounts.Contact attribute), 33
address (dyn.tm.accounts.User attribute), 27
address (dyn.tm.records.AAAARecord attribute), 36
address (dyn.tm.records.ARecord attribute), 36
address (dyn.tm.services.active_failover.ActiveFailover attribute), 56
address (dyn.tm.services.ddns.DynamicDNS attribute), 58
address (dyn.tm.services.gslb.GSLBRegionPoolEntry attribute), 101
address (dyn.tm.services.rttm.RegionPoolEntry attribute), 109
address_2 (dyn.tm.accounts.Contact attribute), 33
address_2 (dyn.tm.accounts.User attribute), 27
AESCipher (class in dyn.encrypt), 129
algorithm (dyn.tm.records.CDNSKEYRecord attribute), 39
algorithm (dyn.tm.records.CDSRecord attribute), 40
algorithm (dyn.tm.records.CERTRecord attribute), 38
algorithm (dyn.tm.records.DNSKEYRecord attribute), 42
algorithm (dyn.tm.records.DSRecord attribute), 43
algorithm (dyn.tm.records.IPSECKEYRecord attribute), 46
algorithm (dyn.tm.records.KEYRecord attribute), 44
alias (dyn.tm.records.ALIASRecord attribute), 37
ALIASRecord (class in dyn.tm.records), 37
all_failover_chains (dyn.tm.services.dsf.TrafficDirector attribute), 94
all_record_sets (dyn.tm.services.dsf.TrafficDirector attribute), 94
all_records (dyn.tm.services.dsf.TrafficDirector attribute), 94
all_response_pools (dyn.tm.services.dsf.TrafficDirector attribute), 94
all_rulesets (dyn.tm.services.dsf.TrafficDirector attribute), 94
all_users (dyn.tm.accounts.PermissionsGroup attribute), 30
altitude (dyn.tm.records.LOCRecord attribute), 45
apmc (dyn.tm.services.rttm.RTTMRegion attribute), 110
ARecord (class in dyn.tm.records), 36
authenticate() (dyn.tm.session.DynectSession method), 12
auto_recover (dyn.tm.services.active_failover.ActiveFailover attribute), 56
auto_recover (dyn.tm.services.gslb.GSLB attribute), 104
auto_recover (dyn.tm.services.rttm.RTTM attribute), 112
automation (dyn.tm.services.dsf.DSFRecordSet attribute), 77
automation (dyn.tm.services.dsf.DSFResponsePool attribute), 83
autopopulate (dyn.tm.services.rttm.RTTMRegion attribute), 110

B

block() (dyn.tm.accounts.UpdateUser method), 24
block() (dyn.tm.accounts.User method), 27

C

CAARecord (class in dyn.tm.records), 37
CDNSKEYRecord (class in dyn.tm.records), 39
CDSRecord (class in dyn.tm.records), 40
cert_usage (dyn.tm.records.TLSARecord attribute), 52
certificate (dyn.tm.records.CERTRecord attribute), 38
certificate (dyn.tm.records.TLSARecord attribute), 52
CERTRecord (class in dyn.tm.records), 38
change_ip() (in module dyn.tm.tools), 115
city (dyn.tm.accounts.Contact attribute), 34
city (dyn.tm.accounts.User attribute), 27
close_session() (dyn.core.SessionEngine class method), 132
cname (dyn.tm.records.CNAMERecord attribute), 41
CNAMERecord (class in dyn.tm.records), 41
connect() (dyn.core.SessionEngine method), 132
Contact (class in dyn.tm.accounts), 33
contact (dyn.tm.zones.Zone attribute), 15
contact_nickname (dyn.tm.services.active_failover.ActiveFailover attribute), 56
contact_nickname (dyn.tm.services.dnssec.DNSSEC attribute), 61
contact_nickname (dyn.tm.services.gslb.GSLB attribute), 104
contact_nickname (dyn.tm.services.rttm.RTTM attribute), 112
contact_nickname (dyn.tm.zones.SecondaryZone attribute), 19
core (dyn.tm.services.dsf.DSFFailoverChain attribute), 81
core_set_count (dyn.tm.services.dsf.DSFResponsePool attribute), 83
country (dyn.tm.accounts.Contact attribute), 34
country (dyn.tm.accounts.User attribute), 27
create() (dyn.tm.services.dsf.DSFResponsePool method), 83
create() (dyn.tm.services.dsf.DSFRuleset method), 86
criteria (dyn.tm.services.dsf.DSFRuleset attribute), 86
criteria_type (dyn.tm.services.dsf.DSFRuleset attribute), 86
CSYNCRecord (class in dyn.tm.records), 39

D

deactivate() (dyn.tm.services.active_failover.ActiveFailover method), 56
deactivate() (dyn.tm.services.ddns.DynamicDNS method), 58
deactivate() (dyn.tm.services.dnssec.DNSSEC method), 61
deactivate() (dyn.tm.services.gslb.GSLB method), 104
deactivate() (dyn.tm.services.reversedns.ReverseDNS method), 106
deactivate() (dyn.tm.services.rttm.RTTM method), 112
deactivate() (dyn.tm.zones.SecondaryZone method), 19
decrypt() (dyn.encrypt.AESCipher method), 129
del_notifier() (dyn.tm.services.dsf.TrafficDirector method), 94
del_recipient() (dyn.tm.services.dsf.DSFNotifier method), 92
delete() (dyn.tm.accounts.Contact method), 34
delete() (dyn.tm.accounts.Notifier method), 32
delete() (dyn.tm.accounts.PermissionsGroup method), 30
delete() (dyn.tm.accounts.UpdateUser method), 24
delete() (dyn.tm.accounts.User method), 27
delete() (dyn.tm.records.DNSRecord method), 35
delete() (dyn.tm.records.SOARecord method), 50
delete() (dyn.tm.services.active_failover.ActiveFailover method), 56
delete() (dyn.tm.services.ddns.DynamicDNS method), 58
delete() (dyn.tm.services.dnssec.DNSSEC method), 61
delete() (dyn.tm.services.dsf.DSFFailoverChain method), 81
delete() (dyn.tm.services.dsf.DSFMonitor method), 90
delete() (dyn.tm.services.dsf.DSFNotifier method), 92
delete() (dyn.tm.services.dsf.DSFRecordSet method), 77
delete() (dyn.tm.services.dsf.DSFResponsePool method), 84
delete() (dyn.tm.services.dsf.DSFRuleset method), 86
delete() (dyn.tm.services.dsf.TrafficDirector method), 94
delete() (dyn.tm.services.gslb.GSLB method), 104
delete() (dyn.tm.services.gslb.GSLBRegion method), 102
delete() (dyn.tm.services.gslb.GSLBRegionPoolEntry method), 101
delete() (dyn.tm.services.reversedns.ReverseDNS method), 106
delete() (dyn.tm.services.rttm.RegionPoolEntry method), 109
delete() (dyn.tm.services.rttm.RTTM method), 112
delete() (dyn.tm.services.rttm.RTTMRegion method), 110
delete() (dyn.tm.zones.Node method), 21
delete() (dyn.tm.zones.SecondaryZone method), 19
delete() (dyn.tm.zones.Zone method), 15
delete_forbid_rule() (dyn.tm.accounts.User method), 27
delete_permission() (dyn.tm.accounts.User method), 27
delete_permissions_group() (dyn.tm.accounts.User method), 27
delete_subgroup() (dyn.tm.accounts.PermissionsGroup method), 30
delete_zone() (dyn.tm.accounts.User method), 27
description (dyn.tm.accounts.PermissionsGroup attribute), 30
DHCIDRecord (class in dyn.tm.records), 41
digest (dyn.tm.records.CDSRecord attribute), 40
digest (dyn.tm.records.DHCIDRecord attribute), 41
digest (dyn.tm.records.DSRecord attribute), 43
digtype (dyn.tm.records.CDSRecord attribute), 40
digtype (dyn.tm.records.DSRecord attribute), 43
dname (dyn.tm.records.DNAMERecord attribute), 42
DNAMERecord (class in dyn.tm.records), 41
DNSKEYRecord (class in dyn.tm.records), 42
DNSRecord (class in dyn.tm.records), 35
DNSSEC (class in dyn.tm.services.dnssec), 60
DNSSECKey (class in dyn.tm.services.dnssec), 60
dsf_id (dyn.tm.services.dsf.DSFFailoverChain attribute), 81
dsf_id (dyn.tm.services.dsf.DSFRecordSet attribute), 78
dsf_id (dyn.tm.services.dsf.DSFResponsePool attribute), 84
dsf_id (dyn.tm.services.dsf.DSFRuleset attribute), 87
dsf_monitor_id (dyn.tm.services.dsf.DSFMonitor attribute), 90
dsf_monitor_id (dyn.tm.services.dsf.DSFRecordSet attribute), 78
dsf_service_ids (dyn.tm.services.dsf.DSFNotifier attribute), 92
DSFAAAARecord (class in dyn.tm.services.dsf), 64
DSFALIASRecord (class in dyn.tm.services.dsf), 64
DSFARecord (class in dyn.tm.services.dsf), 63
DSFCERTRecord (class in dyn.tm.services.dsf), 64
DSFCNAMERecord (class in dyn.tm.services.dsf), 65
DSFDHCIDRecord (class in dyn.tm.services.dsf), 65
DSFDNAMERecord (class in dyn.tm.services.dsf), 66
DSFDNSKEYRecord (class in dyn.tm.services.dsf), 66
DSFDSRecord (class in dyn.tm.services.dsf), 67
DSFFailoverChain (class in dyn.tm.services.dsf), 80
DSFIPSECKEYRecord (class in dyn.tm.services.dsf), 69
DSFKEYRecord (class in dyn.tm.services.dsf), 68
DSFKXRecord (class in dyn.tm.services.dsf), 68
DSFLOCRecord (class in dyn.tm.services.dsf), 69
DSFMonitor (class in dyn.tm.services.dsf), 90
DSFMXRecord (class in dyn.tm.services.dsf), 70
DSFNAPTRRecord (class in dyn.tm.services.dsf), 71
DSFNotifier (class in dyn.tm.services.dsf), 91
DSFNSAPRecord (class in dyn.tm.services.dsf), 72
DSFNSRecord (class in dyn.tm.services.dsf), 73
DSFPTRRecord (class in dyn.tm.services.dsf), 71
DSFPXRecord (class in dyn.tm.services.dsf), 72
DSFRecordSet (class in dyn.tm.services.dsf), 77

DSFResponsePool (class in dyn.tm.services.dsf), 83
DSFRPRecord (class in dyn.tm.services.dsf), 73
DSFRuleset (class in dyn.tm.services.dsf), 86
DSFSPFRecord (class in dyn.tm.services.dsf), 74
DSFSRVRecord (class in dyn.tm.services.dsf), 74
DSFSSHFPRecord (class in dyn.tm.services.dsf), 75
DSFTXTRecord (class in dyn.tm.services.dsf), 75
DSRecord (class in dyn.tm.records), 43
dyn.mm.errors (module), 124
dyn.mm.session (module), 124
dyn.tm.errors (module), 117
DynamicDNS (class in dyn.tm.services.ddns), 58
DynectAuthError, 117
DynectCreateError, 117
DynectDeleteError, 118
DynectGetError, 117
DynectInvalidArgumentError, 117
DynectQueryTimeout, 118
DynectSession (class in dyn.tm.session), 11
DynectUpdateError, 117
DynInvalidArgumentError, 124

E

eligible (dyn.tm.services.dsf.DSFRecordSet attribute), 78
eligible (dyn.tm.services.dsf.DSFResponsePool attribute), 84
EMail (class in dyn.mm.message), 121
email (dyn.tm.accounts.Contact attribute), 34
email (dyn.tm.accounts.User attribute), 27
EmailInvalidArgumentError, 125
EmailKeyError, 124
EmailObjectError, 125
encrypt() (dyn.encrypt.AESCipher method), 130
endpoints (dyn.tm.services.dsf.DSFMonitor attribute), 90
ep (dyn.tm.services.rttm.RTTMRegion attribute), 110
epmc (dyn.tm.services.rttm.RTTMRegion attribute), 110
exchange (dyn.tm.records.KXRecord attribute), 44
exchange (dyn.tm.records.MXRecord attribute), 46
execute() (dyn.core.SessionEngine method), 132
expected (dyn.tm.services.active_failover.HealthMonitor attribute), 54
expected (dyn.tm.services.gslb.Monitor attribute), 100
expected (dyn.tm.services.rttm.Monitor attribute), 108

F

fail_count (dyn.tm.services.dsf.DSFRecordSet attribute), 78
failover_chain_id (dyn.tm.services.dsf.DSFFailoverChain attribute), 81
failover_chains (dyn.tm.services.dsf.DSFResponsePool attribute), 84
failover_chains (dyn.tm.services.dsf.TrafficDirector attribute), 94
failover_data (dyn.tm.services.active_failover.ActiveFailover attribute), 56
failover_data (dyn.tm.services.gslb.GSLBRegion attribute), 102
failover_data (dyn.tm.services.rttm.RTTMRegion attribute), 110
failover_mode (dyn.tm.services.active_failover.ActiveFailover attribute), 56
failover_mode (dyn.tm.services.gslb.GSLBRegion attribute), 102
failover_mode (dyn.tm.services.rttm.RTTMRegion attribute), 110
fax (dyn.tm.accounts.Contact attribute), 34
fax (dyn.tm.accounts.User attribute), 27
first_name (dyn.tm.accounts.Contact attribute), 34
first_name (dyn.tm.accounts.User attribute), 27
flags (dyn.tm.records.CAARecord attribute), 38
flags (dyn.tm.records.CDNSKEYRecord attribute), 39
flags (dyn.tm.records.CSYNCRecord attribute), 40
flags (dyn.tm.records.DNSKEYRecord attribute), 42
flags (dyn.tm.records.KEYRecord attribute), 44
flags (dyn.tm.records.NAPTRRecord attribute), 47
forbid (dyn.tm.accounts.User attribute), 27
format (dyn.tm.records.CERTRecord attribute), 38
fqdn (dyn.tm.records.DNSRecord attribute), 35
fqdn (dyn.tm.services.active_failover.ActiveFailover attribute), 56
fqdn (dyn.tm.services.ddns.DynamicDNS attribute), 58
fqdn (dyn.tm.services.gslb.GSLBRegion attribute), 102
fqdn (dyn.tm.services.gslb.GSLBRegionPoolEntry attribute), 101
fqdn (dyn.tm.services.reversedns.ReverseDNS attribute), 106
fqdn (dyn.tm.services.rttm.RegionPoolEntry attribute), 109
fqdn (dyn.tm.zones.Zone attribute), 15
freeze() (dyn.tm.zones.Zone method), 15

G

gatetype (dyn.tm.records.IPSECKEYRecord attribute), 46
gateway (dyn.tm.records.IPSECKEYRecord attribute), 46
geo_node (dyn.tm.records.DNSRecord attribute), 35
geo_rdata (dyn.tm.records.DNSRecord attribute), 35
get_all_active_failovers() (dyn.tm.zones.Zone method), 15
get_all_advanced_redirect() (dyn.tm.zones.Zone method), 16
get_all_ddns() (dyn.tm.zones.Zone method), 16
get_all_dnssec() (in module dyn.tm.services.dnssec), 59
get_all_dsf_monitors() (in module dyn.tm.services.dsf), 62

get_all_dsf_services() (in module dyn.tm.services.dsf), 62
get_all_failover_chains() (in module dyn.tm.services.dsf), 63
get_all_gslb() (dyn.tm.zones.Zone method), 16
get_all_httpredirect() (dyn.tm.zones.Zone method), 16
get_all_nodes() (dyn.tm.zones.Zone method), 16
get_all_notifiers() (in module dyn.tm.services.dsf), 62
get_all_rdns() (dyn.tm.zones.Zone method), 16
get_all_record_sets() (in module dyn.tm.services.dsf), 63
get_all_records() (dyn.tm.zones.Node method), 21
get_all_records() (dyn.tm.zones.Zone method), 16
get_all_records() (in module dyn.tm.services.dsf), 63
get_all_records_by_type() (dyn.tm.zones.Node method), 21
get_all_records_by_type() (dyn.tm.zones.Zone method), 16
get_all_response_pools() (in module dyn.tm.services.dsf), 63
get_all_rttm() (dyn.tm.zones.Zone method), 16
get_all_rulesets() (in module dyn.tm.services.dsf), 63
get_all_secondary_zones() (in module dyn.tm.zones), 14
get_all_zones() (in module dyn.tm.zones), 14
get_any_records() (dyn.tm.zones.Node method), 21
get_any_records() (dyn.tm.zones.Zone method), 16
get_check_permission() (in module dyn.tm.reports), 114
get_contacts() (in module dyn.tm.accounts), 23
get_dnssec_timeline() (in module dyn.tm.reports), 114
get_log_report() (dyn.tm.services.rttm.RTTM method), 112
get_node() (dyn.tm.zones.Zone method), 16
get_notes() (dyn.tm.zones.Zone method), 16
get_notifiers() (in module dyn.tm.accounts), 23
get_permissions_groups() (in module dyn.tm.accounts), 23
get_qps() (dyn.tm.zones.Zone method), 17
get_qps() (in module dyn.tm.reports), 115
get_rrset_report() (dyn.tm.services.rttm.RTTM method), 112
get_rttm_log() (in module dyn.tm.reports), 114
get_rttm_rrset() (in module dyn.tm.reports), 114
get_session() (dyn.core.SessionEngine class method), 132
get_updateusers() (in module dyn.tm.accounts), 23
get_users() (in module dyn.tm.accounts), 23
get_zone_notes() (in module dyn.tm.reports), 115
group_name (dyn.tm.accounts.PermissionsGroup attribute), 30
group_name (dyn.tm.accounts.User attribute), 27
group_type (dyn.tm.accounts.PermissionsGroup attribute), 30
GSLB (class in dyn.tm.services.gslb), 103
GSLBRegion (class in dyn.tm.services.gslb), 102
GSLBRegionPoolEntry (class in dyn.tm.services.gslb), 101

H

header (dyn.tm.services.active_failover.HealthMonitor attribute), 54
header (dyn.tm.services.gslb.Monitor attribute), 100
header (dyn.tm.services.rttm.Monitor attribute), 108
HealthMonitor (class in dyn.tm.services.active_failover), 54
history (dyn.core.SessionEngine attribute), 132
horiz_pre (dyn.tm.records.LOCRecord attribute), 45
host (dyn.tm.services.active_failover.HealthMonitor attribute), 54
host (dyn.tm.services.gslb.Monitor attribute), 100
host (dyn.tm.services.rttm.Monitor attribute), 108
hosts (dyn.tm.services.reversedns.ReverseDNS attribute), 106
HTMLEMail (class in dyn.mm.message), 122
HTMLTemplateEMail (class in dyn.mm.message), 123

I

implicit_publish (dyn.tm.services.dsf.DSFFailoverChain attribute), 81
implicit_publish (dyn.tm.services.dsf.DSFRecordSet attribute), 78
implicit_publish (dyn.tm.services.dsf.DSFResponsePool attribute), 84
implicit_publish (dyn.tm.services.dsf.DSFRuleset attribute), 87
implicit_publish (dyn.tm.services.dsf.TrafficDirector attribute), 94
implicitPublish (dyn.tm.services.dsf.DSFFailoverChain attribute), 81
implicitPublish (dyn.tm.services.dsf.DSFRecordSet attribute), 78
implicitPublish (dyn.tm.services.dsf.DSFResponsePool attribute), 84
implicitPublish (dyn.tm.services.dsf.DSFRuleset attribute), 87
implicitPublish (dyn.tm.services.dsf.TrafficDirector attribute), 94
interval (dyn.tm.services.active_failover.HealthMonitor attribute), 54
interval (dyn.tm.services.gslb.Monitor attribute), 100
interval (dyn.tm.services.rttm.Monitor attribute), 108
IPSECKEYRecord (class in dyn.tm.records), 45
iptrack_id (dyn.tm.services.reversedns.ReverseDNS attribute), 106

K

KEYRecord (class in dyn.tm.records), 43
keys (dyn.tm.services.dnssec.DNSSEC attribute), 61
keytag (dyn.tm.records.CDSRecord attribute), 40

keytag (dyn.tm.records.DSRecord attribute), 43
KXRecord (class in dyn.tm.records), 44

L

label (dyn.tm.accounts.Notifier attribute), 32
label (dyn.tm.services.dsf.DSFFailoverChain attribute), 81
label (dyn.tm.services.dsf.DSFMonitor attribute), 90
label (dyn.tm.services.dsf.DSFNotifier attribute), 92
label (dyn.tm.services.dsf.DSFRecordSet attribute), 78
label (dyn.tm.services.dsf.DSFResponsePool attribute), 84
label (dyn.tm.services.dsf.DSFRuleset attribute), 87
label (dyn.tm.services.dsf.TrafficDirector attribute), 94
label (dyn.tm.services.gslb.GSLBRegionPoolEntry attribute), 101
label (dyn.tm.services.rttm.RegionPoolEntry attribute), 109
last_name (dyn.tm.accounts.Contact attribute), 34
last_name (dyn.tm.accounts.User attribute), 27
latitude (dyn.tm.records.LOCRecord attribute), 45
link_id (dyn.tm.services.dsf.DSFNotifier attribute), 92
LOCRecord (class in dyn.tm.records), 45
log_out() (dyn.tm.session.DynectSession method), 12
logs (dyn.tm.services.rttm.RegionPoolEntry attribute), 109
longitude (dyn.tm.records.LOCRecord attribute), 45

M

map822 (dyn.tm.records.PXRecord attribute), 48
map_ips() (in module dyn.tm.tools), 116
mapx400 (dyn.tm.records.PXRecord attribute), 48
masters (dyn.tm.zones.SecondaryZone attribute), 19
match_type (dyn.tm.records.TLSARecord attribute), 52
mbox (dyn.tm.records.RPRecord attribute), 49
minimum (dyn.tm.records.SOARecord attribute), 50
MMSession (class in dyn.mm.session), 124
Monitor (class in dyn.tm.services.gslb), 100
Monitor (class in dyn.tm.services.rttm), 108
monitor (dyn.tm.services.active_failover.ActiveFailover attribute), 56
monitor (dyn.tm.services.gslb.GSLB attribute), 104
monitor (dyn.tm.services.rttm.RTTM attribute), 112
monitor_service_ids (dyn.tm.services.dsf.DSFNotifier attribute), 92
MXRecord (class in dyn.tm.records), 46

N

name (dyn.core.SessionEngine attribute), 132
name (dyn.tm.zones.Zone attribute), 17
NAPTRRecord (class in dyn.tm.records), 47
netmask (dyn.tm.services.reversedns.ReverseDNS attribute), 107
new_session() (dyn.core.SessionEngine class method), 132
nickname (dyn.tm.accounts.Contact attribute), 34
nickname (dyn.tm.accounts.UpdateUser attribute), 24
nickname (dyn.tm.accounts.User attribute), 27
Node (class in dyn.tm.zones), 20
node_objects (dyn.tm.services.dsf.TrafficDirector attribute), 94
nodeObjects (dyn.tm.services.dsf.TrafficDirector attribute), 94
nodes (dyn.tm.services.dsf.TrafficDirector attribute), 94
NoSuchAccountError, 125
Notifier (class in dyn.tm.accounts), 32
notifier_id (dyn.tm.accounts.Notifier attribute), 32
notifiers (dyn.tm.services.dsf.TrafficDirector attribute), 94
notify_email (dyn.tm.accounts.Contact attribute), 34
notify_email (dyn.tm.accounts.User attribute), 28
notify_events (dyn.tm.services.active_failover.ActiveFailover attribute), 56
notify_events (dyn.tm.services.dnssec.DNSSEC attribute), 61
notify_events (dyn.tm.services.gslb.GSLB attribute), 104
notify_events (dyn.tm.services.rttm.RTTM attribute), 112
nsap (dyn.tm.records.NSAPRecord attribute), 49
NSAPRecord (class in dyn.tm.records), 49
nsdname (dyn.tm.records.NSRecord attribute), 50
NSRecord (class in dyn.tm.records), 50

O

options (dyn.tm.services.dsf.DSFMonitor attribute), 90
order (dyn.tm.records.NAPTRRecord attribute), 47
order_response_pools() (dyn.tm.services.dsf.DSFRuleset method), 87
order_rulesets() (dyn.tm.services.dsf.TrafficDirector method), 94
organization (dyn.tm.accounts.Contact attribute), 34
organization (dyn.tm.accounts.User attribute), 28

P

pager_email (dyn.tm.accounts.Contact attribute), 34
pager_email (dyn.tm.accounts.User attribute), 28
password (dyn.tm.accounts.UpdateUser attribute), 24
path (dyn.tm.services.active_failover.HealthMonitor attribute), 54
path (dyn.tm.services.gslb.Monitor attribute), 101
path (dyn.tm.services.rttm.Monitor attribute), 108
performance_monitor (dyn.tm.services.rttm.RTTM attribute), 112
permission (dyn.tm.accounts.PermissionsGroup attribute), 30
permission (dyn.tm.accounts.User attribute), 28
permissions (dyn.tm.session.DynectSession attribute), 12
PermissionsGroup (class in dyn.tm.accounts), 29

phone (dyn.tm.accounts.Contact attribute), 34
phone (dyn.tm.accounts.User attribute), 28
poll_response() (dyn.core.SessionEngine method), 132
pool (dyn.tm.services.gslb.GSLBRegion attribute), 102
pool (dyn.tm.services.rttm.RTTMRegion attribute), 110
port (dyn.tm.records.SRVRecord attribute), 51
port (dyn.tm.services.active_failover.HealthMonitor attribute), 55
port (dyn.tm.services.gslb.Monitor attribute), 101
port (dyn.tm.services.rttm.Monitor attribute), 108
post_code (dyn.tm.accounts.Contact attribute), 34
post_code (dyn.tm.accounts.User attribute), 28
precedence (dyn.tm.records.IPSECKEYRecord attribute), 46
preference (dyn.tm.records.KXRecord attribute), 44
preference (dyn.tm.records.MXRecord attribute), 47
preference (dyn.tm.records.NAPTRRecord attribute), 47
preference (dyn.tm.records.PXRecord attribute), 48
priority (dyn.tm.records.SRVRecord attribute), 51
probe_interval (dyn.tm.services.dsf.DSFMonitor attribute), 90
protocol (dyn.tm.records.CDNSKEYRecord attribute), 39
protocol (dyn.tm.records.DNSKEYRecord attribute), 42
protocol (dyn.tm.records.KEYRecord attribute), 44
protocol (dyn.tm.services.active_failover.HealthMonitor attribute), 55
protocol (dyn.tm.services.dsf.DSFMonitor attribute), 90
protocol (dyn.tm.services.gslb.Monitor attribute), 101
protocol (dyn.tm.services.rttm.Monitor attribute), 108
ptrdname (dyn.tm.records.PTRRecord attribute), 48
PTRRecord (class in dyn.tm.records), 48
public_key (dyn.tm.records.CDNSKEYRecord attribute), 39
public_key (dyn.tm.records.DNSKEYRecord attribute), 42
public_key (dyn.tm.records.IPSECKEYRecord attribute), 46
public_key (dyn.tm.records.KEYRecord attribute), 44
publish() (dyn.tm.services.dsf.DSFFailoverChain method), 81
publish() (dyn.tm.services.dsf.DSFRecordSet method), 78
publish() (dyn.tm.services.dsf.DSFResponsePool method), 84
publish() (dyn.tm.services.dsf.DSFRuleset method), 87
publish() (dyn.tm.services.dsf.TrafficDirector method), 94
publish() (dyn.tm.zones.Zone method), 17
publish_note (dyn.tm.services.dsf.DSFFailoverChain attribute), 81
publish_note (dyn.tm.services.dsf.DSFRecordSet attribute), 78
publish_note (dyn.tm.services.dsf.DSFResponsePool attribute), 84
publish_note (dyn.tm.services.dsf.DSFRuleset attribute), 87
publish_note (dyn.tm.services.dsf.TrafficDirector attribute), 95
PXRecord (class in dyn.tm.records), 48

R

rdata() (dyn.tm.records.AAAARecord method), 36
rdata() (dyn.tm.records.ALIASRecord method), 37
rdata() (dyn.tm.records.ARecord method), 36
rdata() (dyn.tm.records.CAARecord method), 38
rdata() (dyn.tm.records.CDNSKEYRecord method), 39
rdata() (dyn.tm.records.CDSRecord method), 40
rdata() (dyn.tm.records.CERTRecord method), 38
rdata() (dyn.tm.records.CNAMERecord method), 41
rdata() (dyn.tm.records.CSYNCRecord method), 40
rdata() (dyn.tm.records.DHCIDRecord method), 41
rdata() (dyn.tm.records.DNAMERecord method), 42
rdata() (dyn.tm.records.DNSKEYRecord method), 42
rdata() (dyn.tm.records.DNSRecord method), 35
rdata() (dyn.tm.records.DSRecord method), 43
rdata() (dyn.tm.records.IPSECKEYRecord method), 46
rdata() (dyn.tm.records.KEYRecord method), 44
rdata() (dyn.tm.records.KXRecord method), 44
rdata() (dyn.tm.records.LOCRecord method), 45
rdata() (dyn.tm.records.MXRecord method), 47
rdata() (dyn.tm.records.NAPTRRecord method), 47
rdata() (dyn.tm.records.NSAPRecord method), 49
rdata() (dyn.tm.records.NSRecord method), 50
rdata() (dyn.tm.records.PTRRecord method), 48
rdata() (dyn.tm.records.PXRecord method), 48
rdata() (dyn.tm.records.RPRecord method), 49
rdata() (dyn.tm.records.SOARecord method), 50
rdata() (dyn.tm.records.SPFRecord method), 51
rdata() (dyn.tm.records.SRVRecord method), 51
rdata() (dyn.tm.records.TLSARecord method), 52
rdata() (dyn.tm.records.TXTRecord method), 53
rdata_class (dyn.tm.services.dsf.DSFRecordSet attribute), 78
rec_name (dyn.tm.records.DNSRecord attribute), 35
recipients (dyn.tm.accounts.Notifier attribute), 32
recipients (dyn.tm.services.dsf.DSFNotifier attribute), 92
record_id (dyn.tm.records.DNSRecord attribute), 36
record_set_id (dyn.tm.services.dsf.DSFRecordSet attribute), 78
record_sets (dyn.tm.services.dsf.DSFFailoverChain attribute), 81
record_sets (dyn.tm.services.dsf.TrafficDirector attribute), 95
record_type (dyn.tm.services.ddns.DynamicDNS attribute), 58

record_types (dyn.tm.services.reversedns.ReverseDNS attribute), 107
records (dyn.tm.services.dsf.DSFRecordSet attribute), 78
records (dyn.tm.services.dsf.TrafficDirector attribute), 95
recover() (dyn.tm.services.active_failover.ActiveFailover method), 56
recover() (dyn.tm.services.gslb.GSLB method), 104
recover() (dyn.tm.services.rttm.RTTM method), 112
recovery_delay (dyn.tm.services.active_failover.ActiveFailover attribute), 56
recovery_delay (dyn.tm.services.gslb.GSLB attribute), 104
recovery_delay (dyn.tm.services.rttm.RTTM attribute), 112
rectypes (dyn.tm.records.CSYNCRecord attribute), 40
refresh() (dyn.tm.services.dsf.DSFFailoverChain method), 81
refresh() (dyn.tm.services.dsf.DSFRecordSet method), 78
refresh() (dyn.tm.services.dsf.DSFResponsePool method), 84
refresh() (dyn.tm.services.dsf.DSFRuleset method), 87
refresh() (dyn.tm.services.dsf.TrafficDirector method), 95
regexp (dyn.tm.records.NAPTRRecord attribute), 47
region (dyn.tm.services.gslb.GSLB attribute), 104
region (dyn.tm.services.rttm.RTTM attribute), 112
region_code (dyn.tm.services.gslb.GSLBRegion attribute), 102
region_code (dyn.tm.services.gslb.GSLBRegionPoolEntry attribute), 101
region_code (dyn.tm.services.rttm.RegionPoolEntry attribute), 109
RegionPoolEntry (class in dyn.tm.services.rttm), 109
remove_node() (dyn.tm.services.dsf.TrafficDirector method), 95
remove_orphans() (dyn.tm.services.dsf.TrafficDirector method), 95
remove_permission() (dyn.tm.accounts.PermissionsGroup method), 30
remove_response_pool() (dyn.tm.services.dsf.DSFRuleset method), 87
replace_all_rulesets() (dyn.tm.services.dsf.TrafficDirector method), 95
replace_forbid_rules() (dyn.tm.accounts.User method), 28
replace_one_ruleset() (dyn.tm.services.dsf.TrafficDirector method), 95
replace_permission() (dyn.tm.accounts.User method), 28
replace_permissions() (dyn.tm.accounts.PermissionsGroup method), 30
replace_permissions_group() (dyn.tm.accounts.User method), 28
replace_zones() (dyn.tm.accounts.User method), 28
replacement (dyn.tm.records.NAPTRRecord attribute), 47
reset() (dyn.tm.services.ddns.DynamicDNS method), 58
response_count (dyn.tm.services.dsf.DSFMonitor attribute), 90
response_pool_id (dyn.tm.services.dsf.DSFFailoverChain attribute), 81
response_pool_id (dyn.tm.services.dsf.DSFResponsePool attribute), 84
response_pools (dyn.tm.services.dsf.DSFRuleset attribute), 87
response_pools (dyn.tm.services.dsf.TrafficDirector attribute), 95
retransfer() (dyn.tm.zones.SecondaryZone method), 19
retries (dyn.tm.services.active_failover.HealthMonitor attribute), 55
retries (dyn.tm.services.dsf.DSFMonitor attribute), 90
retries (dyn.tm.services.gslb.Monitor attribute), 101
retries (dyn.tm.services.rttm.Monitor attribute), 108
ReverseDNS (class in dyn.tm.services.reversedns), 106
revert_changes() (dyn.tm.services.dsf.TrafficDirector method), 95
rname (dyn.tm.records.SOARecord attribute), 50
RPRecord (class in dyn.tm.records), 49
rs_chains (dyn.tm.services.dsf.DSFResponsePool attribute), 84
RTTM (class in dyn.tm.services.rttm), 111
RTTMRegion (class in dyn.tm.services.rttm), 109
ruleset_id (dyn.tm.services.dsf.DSFRuleset attribute), 87
ruleset_ids (dyn.tm.services.dsf.DSFResponsePool attribute), 84
rulesets (dyn.tm.services.dsf.TrafficDirector attribute), 95

S

SecondaryZone (class in dyn.tm.zones), 19
selector (dyn.tm.records.TLSARecord attribute), 52
send() (dyn.mm.message.EMail method), 122
send() (dyn.mm.message.HTMLEMail method), 122
send() (dyn.mm.message.HTMLTemplateEMail method), 123
send() (dyn.mm.message.TemplateEMail method), 122
send_command() (dyn.core.SessionEngine method), 132
send_message() (in module dyn.mm.message), 121
serial (dyn.tm.zones.SecondaryZone attribute), 19
serial (dyn.tm.zones.Zone attribute), 17
serial_style (dyn.tm.records.SOARecord attribute), 50
serial_style (dyn.tm.zones.Zone attribute), 17
serve_count (dyn.tm.services.dsf.DSFRecordSet attribute), 78
serve_count (dyn.tm.services.gslb.GSLBRegion attribute), 103
serve_count (dyn.tm.services.rttm.RTTMRegion attribute), 110
serve_mode (dyn.tm.services.gslb.GSLBRegionPoolEntry attribute), 102

serve_mode (dyn.tm.services.rttm.RegionPoolEntry attribute), 109
service_class (dyn.tm.records.NSRecord attribute), 50
service_id (dyn.tm.services.dsf.TrafficDirector attribute), 95
services (dyn.tm.accounts.Notifier attribute), 32
services (dyn.tm.records.NAPTRRecord attribute), 48
SessionEngine (class in dyn.core), 131
set_monitor() (dyn.tm.services.dsf.DSFRecordSet method), 78
Singleton (class in dyn.core), 131
size (dyn.tm.records.LOCRecord attribute), 45
soa_serial (dyn.tm.records.CSYNCRecord attribute), 40
SOARecord (class in dyn.tm.records), 50
SPFRecord (class in dyn.tm.records), 51
SRVRecord (class in dyn.tm.records), 51
state (dyn.tm.accounts.Contact attribute), 34
status (dyn.tm.accounts.UpdateUser attribute), 24
status (dyn.tm.accounts.User attribute), 28
status (dyn.tm.services.active_failover.HealthMonitor attribute), 55
status (dyn.tm.services.dsf.DSFRecordSet attribute), 78
status (dyn.tm.services.gslb.GSLB attribute), 104
status (dyn.tm.services.gslb.Monitor attribute), 101
status (dyn.tm.services.rttm.Monitor attribute), 108
status (dyn.tm.services.rttm.RTTM attribute), 112
status (dyn.tm.services.rttm.RTTMRegion attribute), 110
status (dyn.tm.zones.Zone attribute), 17
subgroup (dyn.tm.accounts.PermissionsGroup attribute), 30
sync() (dyn.tm.services.gslb.GSLB method), 104
sync() (dyn.tm.services.gslb.GSLBRegion method), 103
sync() (dyn.tm.services.gslb.GSLBRegionPoolEntry method), 102
sync_password() (dyn.tm.accounts.UpdateUser method), 24
syslog_delivery (dyn.tm.services.active_failover.ActiveFailover attribute), 56
syslog_delivery (dyn.tm.services.gslb.GSLB attribute), 104
syslog_delivery (dyn.tm.services.rttm.RTTM attribute), 112
syslog_facility (dyn.tm.services.active_failover.ActiveFailover attribute), 57
syslog_facility (dyn.tm.services.gslb.GSLB attribute), 104
syslog_facility (dyn.tm.services.rttm.RTTM attribute), 112
syslog_ident (dyn.tm.services.active_failover.ActiveFailover attribute), 57
syslog_ident (dyn.tm.services.gslb.GSLB attribute), 104
syslog_ident (dyn.tm.services.rttm.RTTM attribute), 112
syslog_port (dyn.tm.services.active_failover.ActiveFailover attribute), 57
syslog_port (dyn.tm.services.gslb.GSLB attribute), 104
syslog_port (dyn.tm.services.rttm.RTTM attribute), 112
syslog_probe_format (dyn.tm.services.active_failover.ActiveFailover attribute), 57
syslog_probe_format (dyn.tm.services.gslb.GSLB attribute), 104
syslog_probe_format (dyn.tm.services.rttm.RTTM attribute), 113
syslog_rttm_format (dyn.tm.services.rttm.RTTM attribute), 113
syslog_server (dyn.tm.services.active_failover.ActiveFailover attribute), 57
syslog_server (dyn.tm.services.gslb.GSLB attribute), 104
syslog_server (dyn.tm.services.rttm.RTTM attribute), 113
syslog_status_format (dyn.tm.services.active_failover.ActiveFailover attribute), 57
syslog_status_format (dyn.tm.services.gslb.GSLB attribute), 105
syslog_status_format (dyn.tm.services.rttm.RTTM attribute), 113

T

tag (dyn.tm.records.CAARecord attribute), 38
tag (dyn.tm.records.CERTRecord attribute), 38
target (dyn.tm.records.SRVRecord attribute), 51
task (dyn.tm.services.active_failover.ActiveFailover attribute), 57
task (dyn.tm.services.gslb.GSLB attribute), 105
task (dyn.tm.services.gslb.GSLBRegion attribute), 103
task (dyn.tm.services.gslb.GSLBRegionPoolEntry attribute), 102
task (dyn.tm.services.rttm.RegionPoolEntry attribute), 109
task (dyn.tm.services.rttm.RTTM attribute), 113
task (dyn.tm.services.rttm.RTTMRegion attribute), 110
task (dyn.tm.zones.SecondaryZone attribute), 19
task (dyn.tm.zones.Zone attribute), 17
TemplateEMail (class in dyn.mm.message), 122
thaw() (dyn.tm.zones.Zone method), 17
timeline_report() (dyn.tm.services.dnssec.DNSSEC method), 61
timeout (dyn.tm.services.active_failover.HealthMonitor attribute), 55
timeout (dyn.tm.services.gslb.Monitor attribute), 101
timeout (dyn.tm.services.rttm.Monitor attribute), 108
TLSARecord (class in dyn.tm.records), 52
to_json() (dyn.tm.services.active_failover.HealthMonitor method), 55
to_json() (dyn.tm.services.dsf.DSFFailoverChain method), 81
to_json() (dyn.tm.services.dsf.DSFNotifier method), 92
to_json() (dyn.tm.services.dsf.DSFRecordSet method), 78

to_json() (dyn.tm.services.dsf.DSFResponsePool method), 84
to_json() (dyn.tm.services.gslb.GSLBRegionPoolEntry method), 102
to_json() (dyn.tm.services.gslb.Monitor method), 101
to_json() (dyn.tm.services.rttm.Monitor method), 108
to_json() (dyn.tm.services.rttm.RegionPoolEntry method), 109
TrafficDirector (class in dyn.tm.services.dsf), 93
trouble_count (dyn.tm.services.dsf.DSFRecordSet attribute), 78
tsig_key_name (dyn.tm.zones.SecondaryZone attribute), 19
ttl (dyn.tm.records.DNSRecord attribute), 36
ttl (dyn.tm.records.SOARecord attribute), 50
ttl (dyn.tm.services.active_failover.ActiveFailover attribute), 57
ttl (dyn.tm.services.dsf.DSFRecordSet attribute), 78
ttl (dyn.tm.services.dsf.TrafficDirector attribute), 95
ttl (dyn.tm.services.gslb.GSLB attribute), 105
ttl (dyn.tm.services.reversedns.ReverseDNS attribute), 107
ttl (dyn.tm.services.rttm.RTTM attribute), 113
ttl (dyn.tm.zones.Zone attribute), 17
txtdata (dyn.tm.records.SPFRecord attribute), 51
txtdata (dyn.tm.records.TXTRecord attribute), 53
txtdname (dyn.tm.records.RPRecord attribute), 49
TXTRecord (class in dyn.tm.records), 52

U

unblock() (dyn.tm.accounts.UpdateUser method), 24
unblock() (dyn.tm.accounts.User method), 28
update_password() (dyn.tm.session.DynectSession method), 12
update_subgroup() (dyn.tm.accounts.PermissionsGroup method), 31
UpdateUser (class in dyn.tm.accounts), 24
uri (dyn.mm.message.EMail attribute), 122
uri_root (dyn.core.SessionEngine attribute), 133
uri_root (dyn.mm.session.MMSession attribute), 124
uri_root (dyn.tm.session.DynectSession attribute), 12
User (class in dyn.tm.accounts), 25
user (dyn.tm.services.ddns.DynamicDNS attribute), 58
user_name (dyn.tm.accounts.PermissionsGroup attribute), 31
user_name (dyn.tm.accounts.UpdateUser attribute), 24
user_name (dyn.tm.accounts.User attribute), 28
user_permissions_report() (dyn.tm.session.DynectSession method), 12

V

value (dyn.tm.records.CAARecord attribute), 38
version (dyn.tm.records.LOCRecord attribute), 45
vert_pre (dyn.tm.records.LOCRecord attribute), 45

W

wait_for_job_to_complete() (dyn.core.SessionEngine method), 133
website (dyn.tm.accounts.Contact attribute), 34
website (dyn.tm.accounts.User attribute), 28
weight (dyn.tm.records.SRVRecord attribute), 52
weight (dyn.tm.services.gslb.GSLBRegionPoolEntry attribute), 102
weight (dyn.tm.services.rttm.RegionPoolEntry attribute), 109

Z

Zone (class in dyn.tm.zones), 14
zone (dyn.tm.accounts.PermissionsGroup attribute), 31
zone (dyn.tm.accounts.User attribute), 28
zone (dyn.tm.records.DNSRecord attribute), 36
zone (dyn.tm.services.active_failover.ActiveFailover attribute), 57
zone (dyn.tm.services.ddns.DynamicDNS attribute), 58
zone (dyn.tm.services.dnssec.DNSSEC attribute), 61
zone (dyn.tm.services.gslb.GSLBRegion attribute), 103
zone (dyn.tm.services.gslb.GSLBRegionPoolEntry attribute), 102
zone (dyn.tm.services.reversedns.ReverseDNS attribute), 107
zone (dyn.tm.services.rttm.RegionPoolEntry attribute), 109
zone (dyn.tm.zones.SecondaryZone attribute), 19