GREASE Documentation

Release 2.1.0

James E. Bell Jr.

May 07, 2019

Contents:

Welcome to GREASE. Target's operations automation platform! Here lies all the documentation on the project! Make sure to checkout the *GREASE Documentation Guide* for more information! Be sure if you're a new user to checkout the User Guide: *The GREASE User Guide*

CHAPTER 1

GREASE Documentation Guide

GREASE is installable as a stand-alone package *or* used as a dependency in your own solutions! Checkout the links below for more information on how to get started.

1.1 What is GREASE

GREASE is a distributed system written in Python designed to automate automation in operations. Typically for an operations engineer to automated a repeated task or recovery they must account for

1. Determine what the issue is
2. Determine what the recovery is
3. Figuring out when to run their recovery
4. Writing the actual recovery
5. Reporting the recovery event to alert developers an issue occurred

Here at Target GRE we noticed that engineers really only cared to actually fix the issue, not all the other stuff that goes along with setting up cron jobs or scheduled tasks, and decided we could do something to make operations easier to automate. GREASE was our solution. In GREASE an operations engineer's workflow looks like this:

1. Determine what the issue is
2. Determine what the recovery is
3. Write a GREASE configuration
4. Write the recovery

1.1.1 What Problem Does it Solve?

GREASE drastically reduces the complexity for our engineers since they can focus on recovery efforts rather than patching their cron servers or "babysitting" their scripts directory. Additionally GREASE abstracts and reduces the amount of code being written over and over to monitor our endpoints and detect when an issue is occurring.

With GREASE engineers can engineer solutions to technical problems, not entire automation solutions

1.1.2 What Does it Mean to be Distributed?

A distributed system is where multiple discrete nodes in a network can communicate with each other to process data in a faster more scalable way. GREASE takes advantage of this design to be able to handle production work loads of modern operations staff.

This also means it is inherently more complex than a traditional application. Don't be worried though, it's simple enough to get up and running! Typically GREASE is run on multiple VM's, Kubernetes Pods or physical hosts, although the entire system can be sustained on a single system for either development or production. *It is recommended in production to have at least two of each prototype (discussed later) to ensure minimum amounts of fault protection*

1.1.3 Where Can it Run?

Pretty much anywhere! GREASE is built in Python (supporting 2.7+ including 3!). This means anything that runs python can run GREASE. The Daemon, our way of running GREASE services, is also cross-platform! We support all Systemd compliant platforms (Linux), MacOS, and Windows Services. Checkout the *GREASE Administration: Up & Running* guide for more information on installing it.

1.2 The GREASE User Guide

1.2.1 1. Writing New Configurations

Typically a user will write prototype configurations, so we will focus on those here. To learn more about all the different types of configurations see the *The GREASE Data Models* page for more information.

A Prototype Config tells GREASE where to look, what to watch for , where to run, and what to pass to a command. The schema for a configuration can be found here *The GREASE Data Models*. Let's say you have GREASE installed and you want to monitor `localhost` to make sure a webserver is still running via a synthetic transaction using a GET request. That configuration would look something like this:

```
{
    "name": "webserver_check_alive",
    "job": "reboot_local_webserver",
    "source": "url_source",
    "url": ["http://localhost:3000/users/7"],
    'minute': 30,
    "logic": {
        "Range": [
            {
                "field": "status_code",
                "min": 200,
                "max": 200,
                "variable": true,
                "variable_name": "status_code"
            }
        ]
    }
}
```

This configuration will tell GREASE to perform a HTTP GET on the address `http://localhost:3000/users/7` once every hour and ensure that a status code of 200 is returned. If it is not then the job

`reboot_local_webserver` will be scheduled to run. It will be passed the variable `status_code` in its' context.

Now that we have a config written lets focus on the command `reboot_local_webserver` in the next section.

1.2.2 2. Writing Commands

Commands are Python classes that are executed when configurations determine to do so. Building from *1. Writing New Configurations* lets now write the command `reboot_local_webserver`.

Commands extend a base class `tgt_grease.core.Types.Command` found here: *The GREASE Command*. Here is a basic command for explaining a command:

```
from tgt_grease.core.Types import Command
import subprocess

# Class name doesn't have to match command. But your Plugin Package must export the␣
↪matching name (use alias')
class RestartLocalWebServer(Command):

    def __init__(self):
        super(RestartLocalWebServer, self).__init__()

    def execute(self, context):
        # Send a notification
        self.ioc.getNotification().SendMessage("ERROR: Local Web Server returned back␣
↪health check::status code [{0}]".format(context.get("status_code")))
        # Attempt the restart
        return self.restart_nginx()

    def restart_nginx(self):
        if subprocess.call(["systemctl", "restart", "nginx"]) == 0:
            return True
        else:
            return False
```

This command attempts to restart Nginx. If successful it will return true telling the engine the recovery has been performed. If it does not get a good exit code it returns false letting the engine know it needs to attempt the recovery again. **NOTE**: GREASE will only attempt a recovery 6 times before determining the command cannot succeed and stops attempting execution of it.

Since this is just traditional Python code you can do anything here! Call PowerShell scripts, interact with executables, call some Java Jar you need, the possibilities are endless. All the GREASE Prototypes extend this base class and run all of what we call GREASE.

Variable Storage

Lets say we want to expand this example. We now want to execute only on the fifth 404. Rather than making the configuration language very complicated we chose to put application logic in the applications, or as we call them, commands. This is where variable storage comes into play. Since commands are typically short lived executions we offer the `variable_storage` property to your commands which is a provisioned collection just for your command. Lets refactor our command to use this feature:

```
from tgt_grease.core.Types import Command
import subprocess
import datetime
```

(continues on next page)

(continued from previous page)

```
# Class name doesn't have to match command. But your Plugin Package must export the␣
↪matching name (use alias')
class RestartLocalWebServer(Command):

    def __init__(self):
        super(RestartLocalWebServer, self).__init__()

    def execute(self, context):
        # Store the new bad status code
        self.variable_storage.insert_one(
            {
                'createTime': (datetime.datetime.utcnow() + datetime.
↪timedelta(hours=6)),
                'docType': 'statusCode',
                'status_code': context.get('status_code')
            }
        )
        # Ensure a TTL for these documents
        self.variable_storage.create_index([('createTime', 1), ('expireAfterSeconds',␣
↪1)])
        # Count the number of bad status'
        if self.variable_storage.find({'docType': 'status_code'}).count() > 5:
            # Send a notification
            self.ioc.getNotification().SendMessage("ERROR: Local Web Server returned␣
↪back health check::status code [{0}]".format(context.get("status_code")))
            # Attempt the restart
            return self.restart_nginx()
        else:
            # Don't have enough bad status codes yet, so no failure condition
            return True

    def restart_nginx(self):
        if subprocess.call(["systemctl", "restart", "nginx"]) == 0:
            return True
        else:
            return False
```

Look at that! We have a pretty complete program there to restart a web server in the event of more than 5 bad requests sent to a web server.

1.2.3 3. Testing your Code

Testing your code is very important, especially when you are engineering reliability! So GREASE helps with that too! Using the built in test class found here: *The Automation Tester*. Lets continue our series by writing a test for our command. First We will write a test using the original command without Variable Storage:

```
from tgt_grease.core.Types import AutomationTest
from tgt_grease.core import Configuration
from my_demo_package import webserver_check_alive


class TestLocalWSRestart(AutomationTest):

    def __init__(self, *args, **kwargs):
```

(continues on next page)

(continued from previous page)

```
        AutomationTest.__init__(self, *args, **kwargs)
        # For example our file name is basic.config.json so after install it will be
↪here
        self.configuration = "fs://{0}".format(Configuration.greaseDir + 'etc/basic.
↪config.json')
        # Mock data we expect to see from the web server
        self.mock_data = {
            'url': 'localhost:8000',
            'status_code': 500
        }
        # What we expect detection to tell us
        self.expected_data = {
            'status_code': 500
        }
        # Enable testing
        self.enabled = True

    def test_command(self):
        d = webserver_check_alive()
        # Overload the restart_nginx function for testing purposes
        d.restart_nginx = lambda: True
        self.assertTrue(d.execute({'status_code': 500}))
```

Now when running *python setup.py test* on your plugin your commands will be tested for their ability to detect correctly and execute the way you would like. Since we use standard tooling you can also use tools to extract code coverage and other statistics.

Testing with Variable Storage

Testing commands that use Variable Storage is just as simple. We just need to refactor our test a little bit to arrange state around the command a bit:

```
from tgt_grease.core.Types import AutomationTest
from tgt_grease.core import Configuration
from my_demo_package import webserver_check_alive
import datetime


class TestLocalWSRestart(AutomationTest):

    def __init__(self, *args, **kwargs):
        AutomationTest.__init__(self, *args, **kwargs)
        # For example our file name is basic.config.json so after install it will be
↪here
        self.configuration = "fs://{0}".format(Configuration.greaseDir + 'etc/basic.
↪config.json')
        # Mock data we expect to see from the web server
        self.mock_data = {
            'url': 'localhost:8000',
            'status_code': 500
        }
        # What we expect detection to tell us
        self.expected_data = {
            'status_code': 500
        }
```

(continues on next page)

(continued from previous page)

```
        # Enable testing
        self.enabled = True

    def test_command(self):
        d = webserver_check_alive()
        # Create some state in the database
        for i in range(0, 5):
            d.variable_storage.insert_one(
                {
                    'createTime': (datetime.datetime.utcnow() + datetime.
↪timedelta(hours=6)),
                    'docType': 'statusCode',
                    'status_code': context.get('status_code')
                }
            )
        # Overload the restart_nginx function for testing purposes
        d.restart_nginx = lambda: True
        self.assertTrue(d.execute({'status_code': 500}))
        # Now just clean up state
        d.variable_storage.drop()
```

1.3 GREASE Administration: Up & Running

1.3.1 Initial Installation

Via PyPi: The Traditional Way

GREASE is built like any other traditional python software, as a package. This means many things but luckily for the systems administrator it means we traverse the typical PyPi pipeline allowing for you to specify your version and allow for updates to be pretty easy. Here is the installation steps:

1. Run *pip install tgt_grease*
2. Setup your configuration file following

From Source on GitHub: Because you're cool like that

GREASE is developer friendly! We're always looking for new ways to bring joy to our guests as well as our developers. To install GREASE via source follow these steps:

1. Download the repo either via Git or HTTP from GitHub
2. Enter the project folder you created
3. Run *python setup.py install*
    - **NOTE:** We recommend for all use cases to use a virtual environment
4. Setup your configuration file

1.3.2 Understanding & Configuring Your System

There are multiple definitions for configuration in GREASE. The primary ones are:

1. Node Configuration: This refers to the local server's configuration for things like MongoDB credentials & resource limits
2. Cluster Configuration: This refers to the configuration of Job Servers inside of the cluster
3. Prototype Configuration: These are the configurations for prototypes, things such as sourcing & detection

Node Configuration

Node configuration is the local file the running instance uses to execute GREASE operations. It is stored in a file called `grease.conf.json`. This is stored in the GREASE directory. On Unix-like operating systems is found at `/opt/grease/` and for Windows `C:\\grease\\`. You can override this behavior using the environment variable `GREASE_DIR`.

The default configuration looks like this:

```
{
    'Connectivity': {
        'MongoDB': {
            'host': 'localhost',
            'port': 27017
        }
    },
    'Logging': {
        'mode': 'filesystem',
        'verbose': False,
        'trace': False,
        'foreground': False,
        'file': Configuration.greaseDir + 'log' + os.sep + 'grease.log'
    },
    'Notifications': {
        'HipChat': {
            'enabled': False,
            'token': None,
            'room': None
        }
    },
    'Configuration': {
        'dir': Configuration.greaseDir + 'etc' + os.sep
    },
    'Sourcing': {
        'dir': Configuration.greaseDir + 'etc' + os.sep,
        'source': None,
        'config': None,
        'mock': False
    },
    'Import': {
        'searchPath': [
            'tgt_grease.router.Commands',
            'tgt_grease.enterprise.Prototype',
            'tgt_grease.management.Commands',
            'tgt_grease.enterprise.Sources',
            'tgt_grease.enterprise.Detectors',
            'tgt_grease.core',
            'tgt_grease'
        ]
    },
    "NodeInformation": {
```

(continues on next page)

(continued from previous page)

```
        "ResourceMax": 95,
        "DeduplicationThreads": 150
    },
    "Additional": {}
}
```

Lets go through each key and the properties below, what they control and some values you may want to use.

- **Connectivity: This is the store for details around connectivity**
  - MongoDB: This key is the key used to find connection details about the central database

Key	value type	default
host	str	localhost
port	int	27017
username	str	
password	str	
db	str	

- **Logging: Logging configuration information**
  - mode: only supports filesystem logging currently to the log file
  - verbose: Can either be True or False. Setting it to True would print any message where the verbose flag was passed. Note, the only internal system of GREASE that utilizes verbose is deduplication. The rest is in tracing
  - trace: Can either be True or False. This enables tracing from within GREASE. This will show a "stream of consciousness" in the log files.
  - foreground: Can either be True or False. True would print log messages to stdout as well as a log file
  - file: Log file to write messages to
- Notifications: Stores information about notification channels. All channels will need at least one key, "enabled" with a boolean True/False value to enable or disable the channel. All other keys are dependent on the notification channel
- **Configuration: This section contains information about this node's prototype configurations**
  - dir: A directory string on where to load configurations from
- **Sourcing: This section contains information about this node's sourcing prototype configuration**
  - dir: A directory string on where to load configurations from
  - source: A string defaulted to null that if provided sourcing will focus only on prototype configurations from that source to get source data from
  - config: A string defaulted to null that if provided sourcing will focus only that prototype configuration
  - mock: A boolean value which when enabled will attempt to source mocking data dependent from the prototype configurations
- **Import: This section holds information about the import system**
  - searchPath: A list of strings of packages to attempt loading commands from
- **NodeInformation: This section controls how GREASE performs on the Node**
  - ResourceMax: Integer that GREASE uses to ensure that new jobs or processes are not spun up if *memory or CPU* utilization exceed this limit

  - DeduplicationThreads: This integer is how many threads to keep open at one time during deduplication. On even the largest source data sets the normal open threads is 30 but this provides a safe limit at 150 by default

- Additional: Unused currently but can be used for additional user provided configuration

Cluster Configuration

Cluster configuration is stored in the MongoDB collection JobServer. Check the *The GREASE Data Models* for more information about what is stored here.

Prototype Configuration

Prototype configuration is stored in the MongoDB collection Configuration, in the filesystem or located in the package. Check the *The GREASE Data Models* for more information about what is stored here and the schema.

1.3.3 Installing the Daemon

Installing the GREASE daemon is super easy. **Be sure you are logged in or running a console with administrative privileges**. Now install the daemon *on all supported platforms* by running `grease daemon install`. On Unixlike systems you should now have a Systemd Service installed with the service file being stored at `/etc/systemd/system/grease.service` and for Launchd at `/Library/LaunchDaemons/net.grease.daemon.plist`. For windows you will have a new service installed.

You can now control the operations of your cluster!

1.4 The GREASE Data Models

This section covers the different collections of GREASE's MongoDB instance.

1.4.1 Prototype Configurations (GREASE configs)

Prototype configurations tell GREASE what to do with data it detects, and where to detect it from. A Typical config will look like this:

```
{
    "name": String, # <-- Unique name for your configuration
    "job": String, # <-- name of command to be run if logic is true
    "exe_env": String, # <-- If not provided will be default as 'general'
    "source": String, # <-- source of data to be provided
    "retry_maximum": int, # <-- Maximum number of times your command will run before␣
↪stopping. Default is 5 retries.
    "logic": { # <-- Logical blocks to be evaluated by Detection
        "Regex": [ # <-- example for regex detector
            {
                "field": String, # <-- field to process
                "pattern": String # <-- pattern to match
            }
        ]
    }
}
```

NOTE: This is only an example. See the detection documentation for all the options available to you!

These are stored as JSON files either in the config directory of the project `<PROJECT_ROOT>/tgt_grease/enterprise/Model/config` or in the GREASE directory in the etc folder `<GREASE_DIR>/etc/` and the file ends with `.config.json`. Another place to store these configurations is within MongoDB in the `Configuration` collection with the key `type` set to `prototype_config`.

1.4.2 MongoDB

The JobServer Collection

The JobServer collection is used for data pertaining to Job Servers/Nodes of a GREASE cluster. A typical record looks like this:

```
{
    '_id': ObjectId, # <-- MongoDB ID
    'jobs': Int, # <-- Amount of jobs a node has been assigned
    'os': String, # <-- Operating System of a Node
    'roles': List[String], # <-- Execution Environments a Node will be able to process
    'prototypes': List[String], # <-- Prototypes a Node will run
    'active': Boolean, # <-- Node ready for jobs state
    'activationTime': DateTime # <-- Node activation Time
}
```

This is the central registration of a node in GREASE. A node's registration on their filesystem is the MongoDB ID found in the database.

The SourceData Collection

This collection is responsible for storing all source data. Data is transformed after sourcing traversing through detection & scheduling eventually making it to the JobQueue collection. This is the primary data model found in this collection is this::

```
{
    'grease_data': { # <-- Tracing Data for the object moving through the system
        'sourcing': { # <-- Sourcing Data
            'server': ObjectId # <-- Server the source came from
        },
        'detection': { # <-- Detection Data
            'server': ObjectId, # <-- Server assigned to detection
            'start': DateTime, # <-- Time detection server started detection
            'end': DateTime, # <-- Time detection server completed detection
            'detection': Dict # <-- Fields set to be variables in context if any
        },
        'scheduling': { # <-- Scheduling Data
            'server': ObjectId, # <-- Server assigned to scheduling
            'start': DateTime, # <-- Time scheduling started
            'end': DateTime # <-- Time scheduling completed
        },
        'execution': { # <-- Execution Data
            'server': ObjectId, # <-- Server assigned for execution
            'assignmentTime': DateTime, # <-- Time job was assigned
            'completeTime': DateTime, # <-- Time job was completed
            'executionSuccess': Boolean, # <-- Execution Success
            'commandSuccess': Boolean, # <-- Command Success
```

(continues on next page)

(continued from previous page)

```
            'failures': Int, # <-- Job Execution Failures
            'returnData': Dict # <-- Data returned from command
        }
    },
    'source': String, # <-- source data came from
    'configuration': String, # <-- Name of configuration data came from
    'data': Dict, # <-- Actual Response Data
    'createTime': DateTime, # <-- Time of Entry
    'expiry': DateTime # <-- Expiration time
}
```

Data is collected from a source and distributed to each individual dictionary in the collection. Nodes will pick up each piece of data and process it based on their assignment.

CHAPTER 2

Code Documentation

2.1 GREASE Core

making operations automation automate-able

2.1.1 Subpackages

GREASE Core Types

The GREASE Command

class tgt_grease.core.Types.**Command**(*Logger=None*)

Bases: object

Abstract class for commands in GREASE

__metaclass__

Metadata class object

Type ABCMeta

purpose

The purpose of the command

Type str

help

Help string for the command line

Type str

__author__

Authorship string

Type str

__version__
Command Version

Type str

os_needed
If a specific OS is needed then set this

Type str

ioc
IOC container for access to system resources

Type *GreaseContainer*

variable_storage
collection object for command

Type pymongo.collection

execute(*context*)
Base Execute Method

This method should *always* be overridden in child classes. This is the code that will run when your command is called. If this method is not implemented then the class will fail loading.

Parameters **context** (*dict*) – context for the command to use

Returns Command Success

Return type bool

failures

getData()
Get any data the execute method wanted to put into telemetry

Returns The Key/Value pairs from the execute method execution

Return type dict

getExecVal()
Get the execution attempt success

Returns If the command executed without exception

Return type bool

getRetVal()
Get the execution boolean return state

Returns the boolean return value of execute

Return type bool

help = '\n No Help Information Provided\n '

os_needed = None

prevent_retries()
Sets a flag in the command's return data that will signal to stop retrying, even before the default retry limit is met.

purpose = 'Default'

safe_execute(*context=None*)
Attempt execution and prevent MOST exceptions

Parameters **context** (*dict*) – context for the command to use

Returns Void method to attempt exceptions

Return type None

setData(*Key*, *Data*)
Put Data into the data object to be inserted into telemetry

Parameters

- **Key** (*str*) – Key for the data to be stored
- **Data** (*object*) – JSON-able object to store

Returns Void Method to put data

Return type None

The Scheduled GREASE Command

Note: This command type doesn't need a prototype configuration. Just to be run as a prototype on a GREASE node

class tgt_grease.core.Types.**ScheduledCommand**(*logger=None*)
Bases: tgt_grease.core.Types.Command.Command

Scheduled Commands Run as Prototypes

This type is used for commands that need to be run cyclically. They will be run as prototypes (always running). Make sure to fill out the **timeToRun** and **run** methods

execute(*context*)
Command execute method

This will run continuously waiting for *timeToRun* to return true to call *run*

Parameters **context** (*dict*) – context for the command to use **Not Used Here**

run()
Put your code here to run whenever the conditions in timeToRun are defined

---

Note: We recommend returning something valuable since the engine logs the result of the method in verbose mode

---

timeToRun()
Checks to ensure it is time to run

Returns If time to run then true else false

Return type bool

The Automation Tester

Test your automation configurations and commands with this class! Extend it to get started.

class tgt_grease.core.Types.**AutomationTest**(**args*, ***kwargs*)
Bases: unittest.case.TestCase

Automation Test Class

Version II of GREASE was all about proving stability. Automation testing is critically important to ensure reliability during fault isolation. This class is an abstract class your tests can implement to ensure they will perform exactly as you expect in production.

Make sure you set the **configuration** class attribute to ensure your configuration is tested, the **mock_data** class attribute with your mock data dictionary you expect to be sourced in production, and the **expected_data** with what you expect detection to find from your mocked source data. Then implement the **test_command** method to write standard unittests around your automation. The Platform will test your configuration for you, and execute **test_command** with *python setup.py test* is executed.

configuration
Configuration to load for this test

Type str|dict

mock_data
String Key -> Int/Float/String Value pair to mock source data

Type dict

expected_data
data you expect context for your command to look like

Type dict

enabled
set to true to enable your test to run

Type bool

Here is an example:

```
class TestAutomationTest(AutomationTest):

    def __init__(self, *args, **kwargs):
        AutomationTest.__init__(self, *args, **kwargs)
        self.configuration = "mongo://test_automation_test"
        self.mock_data = {'ver': 'var'}
        self.expected_data = {'ver': ['var']}
        self.enabled = True

    def test_command(self):
        myCommand = myCommand()
        self.assertTrue(myCommand.execute({'hostname': 'localhost'}))
```

This is a pretty basic example but it will help you get started automatically testing your automation!

Note: YOU MUST SET THE PROPERTY 'ENABLED' TO BOOLEAN TRUE IN ORDER FOR YOUR TEST TO BE PICKED UP

Note: To use a static configuration set *configuration* to a dictionary

Note: To use a MongoDB configuration for a test prefix your configuration's name with mongo://

Note: To use a package configuration for a test prefix your configuration's name with pkg://

Note: to use a filesystem configuration for a test prefix your configuration's path with fs://

test_command()
This method is for **you** to fill out to test your command

Note: The more tests the better! Make sure to add as many tests as you need to ensure your automation is always successful

test_configuration()
Configuration Test

This method tests your configuration and validates that detection will return as you expect

2.1.2 Configuration Class

class tgt_grease.core.**Configuration**(*ConfigFile=None*)
Bases: object

GREASE Configuration Management

This class is responsible for management of configuration of a GREASE Node. Default Configuration will be used if the grease.conf.json document is not found in the root of the GREASE directory. It will ensure all folders/files are in the directory and serve as the access point for configuration data

greaseDir
The root directory of GREASE

Type str

fs_sep
The filesystem separator for the installed OS

Type str

greaseConfigFile
Location of the current configuration file

Type str

FileSystem
Directories of the GREASE filesystem

Type list

GREASE_CONFIG
Actual config

Type dict

static DefaultConfig()
Returns the Default GREASE Config

Returns Default Configuration

Return type dict

EnsureGreaseFS()
Ensures the GREASE Directory structure is setup

Returns If the FS is in place then True

Return type bool

FileSystem = ['etc', 'log']

NodeIdentity = 'Unknown'

static ReloadConfig(*ConfigFile=None*)
[Re]loads the configuration

Returns Void Method

Return type None

fs_sep = '/'

static get(*section*, *key=None*, *default=None*)
Retrieve configuration item

Parameters

- **section** (*str*) – Configuration Section to read from
- **key** (*str*) – Configuration key to retrieve
- **default** (*object*) – Default value if section/key is not found

greaseConfigFile = '/opt/grease/grease.conf.json'

greaseDir = '/opt/grease/'

set(*key*, *value*, *section=None*)
Set configuration item

Parameters

- **section** (*str*) – Configuration Section to set
- **key** (*str*) – Configuration key to set
- **value** (*object*) – value to set

Returns Sets item only

Return type None

2.1.3 Mongo Class

class tgt_grease.core.**Mongo**(*Config=None*)
Bases: object

MongoDB Connection Class

_client
The actual PyMongo Connection

Type pymongo.MongoClient

_config
Configuration Object

Type *Configuration*

Client()
get the connection client

Returns Returns the mongoDB connection client

Return type pymongo.MongoClient

Close()
Close PyMongo Connection

Returns Void Method to close connection

Return type None

2.1.4 Importer Class

class tgt_grease.core.**ImportTool**(*logger*)
Bases: object

Import Tooling for getting instances of classes automatically

_log
Logger for the class

Type *Logging*

load(*className*)
Dynamic loading of classes for the system

Parameters **className** (*str*) – Class name to search for

Returns If an object is found it is returned None: If an object is not found and error occurs None is returned

Return type object

2.1.5 GreaseContainer

class tgt_grease.core.**GreaseContainer**(**args*, ***kwargs*)
Bases: object

Inversion of Control Container for objects in GREASE

ensureRegistration()

Returns

getCollection(*collectionName*)
Get a collection object from MongoDB

Parameters **collectionName** (*str*) – Collection to get

Returns Collection instance

Return type pymongo.collection.Collection

getConfig()
Gets the Configuration Instance

Returns the configuration instance

Return type tgt_grease.core.Configuration.Configuration

getLogger()
Get the logging instance

Returns The logging instance

Return type *Logging*

getMongo()
Get the Mongo instance

Returns Mongo Instance Connection

Return type *Mongo*

getNotification()
Get the notifications instance

Returns The notifications instance

Return type *tgt_grease.core.Notifications*

2.1.6 Logging Class

class tgt_grease.core.**Logging**(*Config=None*)
Bases: object

Application Logging for GREASE

This is the primary configuration source for GREASE logging. All log information will be passed here to enable centralized log aggregation

_conf
This is an instance of the Config to enable configuring loggers

Type *Configuration*

_logger
This is the actual logger for GREASE

Type logging.Logger

_formatter
This is the log formatter

Type logging.Formatter

_notifications
Notifications instance

Type *Notifications*

foreground
If set will override config and print all log messages

Type bool

DefaultLogger()
Default Logging Provisioning

Returns void method to provision class internals

Return type None

ProvisionLoggers()
Loads Log Handler & Config

Returns Simple loader, Nothing needed

Return type None

TriageMessage(*message*, *additional=None*, *verbose=False*, *trace=False*, *notify=False*, *level=10*)
Central message handler

Parameters

- **message** (*str*) – Message to Log
- **additional** (*object*) – Additional information to log
- **verbose** (*bool*) – To be printed if verbose is enabled
- **trace** (*bool*) – To be printed if trace is enabled
- **notify** (*bool*) – If true will pass through notification system
- **level** (*int*) – Log Level

Returns Log Success

Return type bool

critical(*message*, *additional=None*, *verbose=False*, *trace=False*, *notify=True*)
Critical Messages

Use this method for logging critical statements

Parameters

- **message** (*str*) – Message to log
- **additional** (*object*) – Additional information to log. Note: object must be able to transform to string
- **verbose** (*bool*) – Print only if verbose mode
- **trace** (*bool*) – Print only if trace mode
- **notify** (*bool*) – Run through the notification management system

Returns Message is logged

Return type bool

debug(*message*, *additional=None*, *verbose=False*, *trace=False*, *notify=False*)
Debug Messages

Use this method for logging debug statements

Parameters

- **message** (*str*) – Message to log
- **additional** (*object*) – Additional information to log. Note: object must be able to transform to string
- **verbose** (*bool*) – Print only if verbose mode
- **trace** (*bool*) – Print only if trace mode
- **notify** (*bool*) – Run through the notification management system

Returns Message is logged

Return type bool

error(*message*, *additional=None*, *verbose=False*, *trace=False*, *notify=True*)
Error Messages

Use this method for logging error statements

Parameters

- **message** (*str*) – Message to log
- **additional** (*object*) – Additional information to log. Note: object must be able to transform to string
- **verbose** (*bool*) – Print only if verbose mode
- **trace** (*bool*) – Print only if trace mode
- **notify** (*bool*) – Run through the notification management system

Returns Message is logged

Return type bool

foreground = False

getConfig()
Getter for Configuration

Returns The loaded configuration object

Return type *Configuration*

getNotification()
Get Notification Class

Returns The current Notifications instance

Return type *Notifications*

info(*message*, *additional=None*, *verbose=False*, *trace=False*, *notify=False*)
Info Messages

Use this method for logging info statements

Parameters

- **message** (*str*) – Message to log
- **additional** (*object*) – Additional information to log. Note: object must be able to transform to string
- **verbose** (*bool*) – Print only if verbose mode
- **trace** (*bool*) – Print only if trace mode
- **notify** (*bool*) – Run through the notification management system

Returns Message is logged

Return type bool

trace(*message*, *additional=None*, *verbose=False*, *trace=True*, *notify=False*)
Trace Messages

Use this method for logging tracing (enhanced debug) statements

Parameters

- **message** (*str*) – Message to log

- **additional** (*object*) – Additional information to log. Note: object must be able to transform to string
- **verbose** (*bool*) – Print only if verbose mode
- **trace** (*bool*) – Print only if trace mode
- **notify** (*bool*) – Run through the notification management system

Returns Message is logged

Return type bool

warning(*message*, *additional=None*, *verbose=False*, *trace=False*, *notify=False*)

Warning Messages

Use this method for logging warning statements

Parameters

- **message** (*str*) – Message to log
- **additional** (*object*) – Additional information to log. Note: object must be able to transform to string
- **verbose** (*bool*) – Print only if verbose mode
- **trace** (*bool*) – Print only if trace mode
- **notify** (*bool*) – Run through the notification management system

Returns Message is logged

Return type bool

2.1.7 Notifier Class

class tgt_grease.core.**Notifications**(*Config=None*)

Bases: object

Notification Router for third party resources

This is the class to handle all notifications to third party resources

_conf

Configuration Object

Type *Configuration*

hipchat_url

This is the hipchat API url

Type str

hipchat_token

set this to override the config for the auth token

Type str

hipchat_room

set this to override the config for the room

Type str

SendMessage(*message*, *level=10*, *channel=None*)
Send Message to configured channels

This method is the main point of contact with Notifications in GREASE. This will handle routing to all configured channels. Use *level* to define what level the message is. This can impact whether a message is sent as well as if the message sent will have special attributes (EX: red text). Use *channel* to route around sending to multiple channels if the message traditionally would go to multiple, instead going only to the selected one.

Note: if you use the channel argument and the channel is not found you will receive False back

Parameters

- **message** (*str*) – Message to send
- **level** (*int*) – Level of message to be sent
- **channel** (*str*) – Specific channel to notify

Returns Success of sending

Return type bool

hipchat_room = None

hipchat_token = None

hipchat_url = 'https://api.hipchat.com/v2/room/'

send_hipchat_message(*message*, *level*, *color=None*)
Send a hipchat message

Parameters

- **message** (*str*) – Message to send to hipchat
- **level** (*int*) – message level
- **color** (*str*) – color of message

Returns API response status

Return type bool

send_slack_message(*message*)
Send a slack message to slack channel using webhook url in the configuration

Parameters **message** (*str*) – Message to send to Slack

Returns API response status

Return type bool

2.2 GREASE Router

2.2.1 Subpackages

Router Commands

Classes

The Daemon Class

class tgt_grease.router.Commands.Daemon.**DaemonProcess**(*ioc*)
Bases: object

Actual daemon processing for GREASE Daemon

ioc
The Grease IOC

Type *GreaseContainer*

current_real_second
Current second in time

Type int

registered
If the node is registered with MongoDB

Type bool

impTool
Instance of Import Tool

Type *ImportTool*

conf
Prototype Configuration Instance

Type *PrototypeConfig*

contextManager = {'jobs': {}, 'prototypes': {}}

current_real_second = None

drain_jobs(*JobCollection*)
Will drain jobs from the current context

This method is used to prevent abnormal ending of executions

Parameters **JobCollection** (*pymongo.collection.Collection*) – Job Collection Object

Returns When job queue is emptied

Return type bool

impTool = None

ioc = None

log_once_per_second(*message*, *level=10*, *additional=None*)
Log Message once per second

Parameters

- **message** (*str*) – Message to log
- **level** (*int*) – Log Level
- **additional** (*object*) – Additional information that is able to be str'd

Returns Void Method to fire log message

Return type None

register()
Attempt to register with MongoDB

Returns Registration Success

Return type bool

registered = True

server()
Server process for ensuring prototypes & jobs are running

By Running this method this will clear the DB of any jobs a node may have

Returns Server Success

Return type bool

Commands

daemon

class tgt_grease.router.Commands.DaemonCmd.**Daemon**
Bases: tgt_grease.core.Types.Command.Command

Daemon Class for the daemon

execute(*context*)
Base Execute Method

This method should *always* be overridden in child classes. This is the code that will run when your command is called. If this method is not implemented then the class will fail loading.

Parameters **context** (*dict*) – context for the command to use

Returns Command Success

Return type bool

help = '\n Provide simple abstraction for daemon operations in GREASE\n \n Args:\n ins

install()
Handle Daemon Installation based on the platform we're working with

Returns installation success

Return type bool

purpose = 'Control Daemon Processing in GREASE'

run(*loop=None*)
Actual running of the daemon

Parameters **loop** (*int*) – Amount of cycles the daemon should run for

Returns Server running state

Return type bool

start()
Starting the daemon based on platform

Returns start success

Return type bool

stop()
Stopping the daemon based on the platform

Returns stop success

Return type bool

help

class tgt_grease.router.Commands.HelpCmd.**Help**
Bases: tgt_grease.core.Types.Command.Command

The Help Command for GREASE

Meant to provide a rich CLI Experience to users to enable quick help

execute(*context*)
Base Execute Method

This method should *always* be overridden in child classes. This is the code that will run when your command is called. If this method is not implemented then the class will fail loading.

Parameters **context** (*dict*) – context for the command to use

Returns Command Success

Return type bool

help = '\n Provide help information to users of GREASE about available commands. This

purpose = 'Provide Help Information'

2.2.2 Classes

2.2.3 Router Class

class tgt_grease.router.**GreaseRouter**
Bases: object

Main GREASE CLI Router

This class handles routing CLI requests as well as starting the Daemon on Windows/POSIX systems

_config
Main Configuration Object

Type *Configuration*

_logger
Main Logging Instance

Type *Logging*

_importTool
Importer Tool Instance

Type *ImportTool*

_exit_message
Exit Message

Type str

StartGREASE()
EntryPoint for CLI scripts for GREASE

Returns Void Method for GREASE

Return type None

exit(*code*, *message=None*)
Exit program with exit code

Parameters

- **code** (*int*) – Exit Code
- **message** (*str*) – Exit message if any

Returns Will exit program

Return type None

get_arguments()
Parse CLI long arguments into dictionaries

This expects arguments separated by space *–opt val*, colon *–opt:val*, or equal *–opt=val* signs

Returns key->value pairs of arguments

Return type object, dict

run()
Route commands through GREASE

Returns Exit Code

Return type int

2.3 GREASE Prototypes

2.3.1 Subpackages

Enterprise Models

The Prototype Configuration Model

class tgt_grease.enterprise.Model.**PrototypeConfig**(*ioc=None*)
Bases: object

Responsible for Scanning/Detection/Scheduling configuration

Structure of Configuration:

```
{
    'configuration': {
        'pkg': [], # <-- Loaded from pkg_resources.resource_filename('tgt_grease.
↪enterprise.Model', 'config/')
        'fs': [], # <-- Loaded from `<GREASE_DIR>/etc/*.config.json`
        'mongo': [] # <-- Loaded from the Configuration Mongo Collection
    },
```

(continues on next page)

(continued from previous page)

```
    'raw': [], # <-- All loaded configurations
    'sources': [], # <-- list of sources found in configurations
    'source': {} # <-- keys will be source values list of configs for that source
    'names': [], # <-- all configs via their name so to allow dialing
    'name': {} # <-- all configs via their name so to allow being dialing
}
```

Structure of a configuration file:

```
{
    "name": String,
    "job": String,
    "exe_env": String, # <-- If not provided will be default as 'general'
    "source": String,
    "logic": {
        # I need to be the logical blocks for Detection
    }
}
```

ioc
IOC access

Type *GreaseContainer*

getConfiguration()
Returns the Configuration Object loaded into memory

Returns Configuration object

Return type dict

get_config(*name*)
Get Configuration by name

Parameters **name** (*str*) – Configuration name to get

Returns Configuration if found else empty dict

Return type dict

get_names()
Returns the list of names of configs

Returns List of config names

Return type list

get_source(*name*)
Get all configuration by source by name

Parameters **name** (*str*) – Source name to get

Returns Configuration if found else empty dict

Return type list[dict]

get_sources()
Returns the list of sources to be scanned

Returns List of sources

Return type list

load(*reloadConf=False*, *ConfigurationList=None*)
[Re]loads configuration data about the current execution node

Configuration data loads from 3 places in GREASE. The first is internal to the package, if one were to manually add their own files into the package in the current directory following the file pattern. The next is following the same pattern but loaded from *<GREASE_DIR>/etc/*. The final place GREASE looks for configuration data is from the *configuration* collection in MongoDB

Parameters

- **reloadConf** (*bool*) – If True this will reload the global object. False will return the object
- **ConfigurationList** (*list of dict*) – If provided will load the list of dict for config after validation

Note: Providing a configuration *automatically* reloads the memory structure of prototype configuration

Returns Current Configuration information

Return type dict

load_from_fs(*directory*)
Loads configurations from provided directory

Note: Pattern is **.config.json*

Parameters **directory** (*str*) – Directory to load from

Returns configurations

Return type list of dict

load_from_mongo()
Returns all active configurations from the mongo collection Configuration

Structure of Configuration expected in Mongo:

```
{
    "name": String,
    "job": String,
    "exe_env": String, # <-- If not provided will be default as 'general'
    "active": Boolean, # <-- set to true to load configuration
    "type": "prototype_config", # <-- MUST BE THIS VALUE; For it is the␣
↪config type :)
    "source": String,
    "logic": {
        # I need to be the logical blocks for Detection
    }
}
```

Returns Configurations

Return type list of dict

validate_config(*config*)
Validates a configuration

The default JSON Schema is this:

```
{
    "name": String,
    "job": String,
    "exe_env": String, # <-- If not provided will be default as 'general'
    "source": String,
    "logic": {
        # I need to be the logical blocks for Detection
    }
}
```

Parameters **config** (*dict*) – Configuration to validate

Returns If it is a valid configuration

Return type bool

validate_config_list(*configs*)
Validates a configuration List

Parameters **configs** (*list[dict]*) – Configuration List

Returns The Valid configurations

Return type list

The Base Source

class tgt_grease.enterprise.Model.**BaseSourceClass**
Bases: object

Base Class for all sources to implement

_data
List of data to be returned to GREASE

Type list[dict]

deduplication_strength
Level of deduplication strength to use *higher is stronger uniqueness*

Type float

field_set
If none all fields found will be duplicated otherwise only fields listed will be

Type None or list

deduplication_expiry
Hours to retain deduplication data

Type int

deduplication_expiry_max
Days to deduplicate for **maximum**

Type int

get_data()
Returns data from source

Returns List of *single dimension* dictionaries for GREASE to parse through other prototypes

Return type list[dict]

mock_data(*configuration*)
Mock the source for data

Use this method to read through configuration provided to you, and mock getting data. This will *always* be called by the scan engine. **Ensure you set any data to the 'self._data' variable. A list of dictionaries for the engine to schedule for detection**

Parameters **configuration** (*dict*) – Configuration for the sourcing to occur with

Note: This is the method to fill out to get data into GREASE.

Returns mock data from source

Return type list[dict]

parse_source(*configuration*)
Parse the source for data

Use this method to read through configuration provided to you, and get data. This will *always* be called by the scan engine. **Ensure you set any data to the 'self._data' variable. A list of dictionaries for the engine to schedule for detection**

Parameters **configuration** (*dict*) – Configuration for the sourcing to occur with

Note: This is the method to fill out to get data into GREASE.

Returns If True data will be scheduled for ingestion after deduplication. If False the engine will bail out

Return type bool

The Base Detector

class tgt_grease.enterprise.Model.**Detector**(*ioc=None*)
Bases: object

Base Detection Class

This is the abstract class for detectors to implement

ioc
IOC Access

Type *GreaseContainer*

processObject(*source*, *ruleConfig*)
Processes an object and returns valid rule data

Data returned in the second parameter from this method should be in this form:

```
{
    '<field>': Object # <-- if specified as a variable then return the key->
↪Value pairs
    ...
}
```

Parameters

- **source** (*dict*) – Source Data
- **ruleConfig** (*list[dict]*) – Rule Configuration Data

Returns first element boolean for success; second dict for any fields returned as variables

Return type tuple

The DeDuplication Engine

class tgt_grease.enterprise.Model.**Deduplication**(*ioc=None*)

Bases: object

Responsible for Deduplication Operations

Deduplication in GREASE is a multi-step process to ensure performance and accuracy of deduplication. The overview of this process is this:

- Step 1: Identify a Object Type 1 Hash Match. A Type 1 Object (T1) is a SHA256 hash of a dictionary in a data list. If we can hash the entire object and find a match then the object is 100% duplicate.
- Step 2: Object Type 2 Matching. If a Type 1 (T1) object cannot be found Type 2 Object (T2) deduplication occurs. This will introspect the dictionary for each field and map them against other likely objects of the same type. If a hash match is found (source + field + value as a SHA256) then the field is 100% duplicate. The aggregate score of all fields or the specified subset is above the provided threshold then the object is duplicate. This prevents similar objects from passing through when they are most likely updates to an original object that does not need to be computed on. If a field updates that you will need always then exclude it will need to be passed into the *Deduplicate* function.

Object examples:

```
# Type 1 Object

{
    '_id': ObjectId, # <-- MongoDB ObjectID
    'type: Int, # <-- Always Type 1
    'hash': String, # <-- SHA256 hash of entire object
    'expiry': DateTime, # <-- Expiration time if no objects are found to be␣
↪duplicate after which object will be deleted
    'max_expiry': DateTime, # <-- Expiration time for object to be deleted when␣
↪reached
    'score': Int, # <-- Amount of times this object has been found
    'source': String # <-- Source of the object
}
# Type 2 Object
{
    '_id': ObjectId, # <-- MongoDB ObjectID
    'type: Int, # <-- Always Type 2
    'source': String, # <-- Source of data
    'field': String, # <-- Field in Object
```

(continues on next page)

(continued from previous page)

```
    'value': String, # <-- Value of Object's field
    'hash': String, # <-- SHA256 of source + field + value
    'expiry': DateTime, # <-- Expiration time if no objects are found to be␣
↪duplicate after which object will be deleted
    'max_expiry': DateTime, # <-- Expiration time for object to be deleted when␣
↪reached
    'score': Int, # <-- Amount of times this object has been found
    'parentId': ObjectId # <-- T1 Object ID from parent
}
```

ioc
IoC access for DeDuplication

Type *GreaseContainer*

Deduplicate(*data*, *source*, *configuration*, *threshold*, *expiry_hours*, *expiry_max*, *collection*, *field_set=None*)
Deduplicate data

This method will deduplicate the *data* object to allow for only unique objects to be returned. The collection variable will be the collection deduplication data will be stored in

Parameters

- **data** (*list[dict]*) – **list or single dimensional dictionaries** to deduplicate
- **source** (*str*) – Source of data being deduplicated
- **configuration** (*str*) – Configuration Name Provided
- **threshold** (*float*) – level of duplication allowed in an object (the lower the threshold the more uniqueness is required)
- **expiry_hours** (*int*) – Hours to retain deduplication data
- **expiry_max** (*int*) – Maximum days to retain deduplication data
- **collection** (*str*) – Deduplication collection to use
- **field_set** (*list, optional*) – Fields to deduplicate on

Note: expiry_hours is specific to how many hours objects will be persisted for if they are not seen again

Returns Deduplicated data

Return type list[dict]

static deduplicate_object(*ioc*, *obj*, *expiry*, *expiry_max*, *threshold*, *source_name*, *configuration_name*, *final*, *collection*, *data_pointer=None*, *data_max=None*, *field_set=None*)
DeDuplicate Object

This is the method to actually deduplicate an object. The *final* argument is appended to with the obj if it was successfully deduplicated.

Parameters

- **ioc** (*GreaseContainer*) – IoC for the instance
- **obj** (*dict*) – Object to be deduplicated

- **expiry** (*int*) – Hours to deduplicate for
- **expiry_max** (*int*) – Maximum days to deduplicate for
- **threshold** (*float*) – level of duplication allowed in an object (the lower the threshold the more uniqueness is required)
- **source_name** (*str*) – Source of data being deduplicated
- **configuration_name** (*str*) – Configuration being deduplicated for
- **final** (*list*) – List to append *obj* to if unique
- **collection** (*str*) – Name of deduplication collection
- **data_pointer** (*int*) – If provided will provide log information relating to thread (Typically used via *Deduplicate*)
- **data_max** (*int*) – If provided will provide log information relating to thread (Typically used via *Deduplicate*)
- **field_set** (*list*) – If provided will only deduplicate on list of fields provided

Returns Nothing returned. Updates *final* object

Return type None

static generate_expiry_time(*hours*)
Generates UTC Timestamp for hours in the future

Parameters **hours** (*int*) – How many hours in the future to expire on

Returns Datetime object for hours in the future

Return type datetime.datetime

static generate_hash_from_obj(*obj*)
Takes an object and generates a SHA256 Hash of it

Parameters **obj** (*object*) – Hashable object ot generate a SHA256

Returns Object Hash

Return type str

static generate_max_expiry_time(*days*)
Generates UTC Timestamp for hours in the future

Parameters **days** (*int*) – How many days in the future to expire on

Returns Datetime object for days in the future

Return type datetime.datetime

static make_hashable(*obj*)
Takes a dictionary and makes a sorted tuple of strings representing flattened key value pairs :param obj: A dictionary :type obj: dict

Returns a sorted flattened tuple of the dictionary's key value pairs

Return type tuple<str>

Example

{ "a": ["test1", "test2"], "b": [{"test2": 21}, {"test1": 1}, {"test7": 3}], "c": "test"

} becomes... (('a', ('test1', 'test2')),

('b', ((('test1', 1),), (('test2', 21),), (('test7', 3),))), ('c', 'test'))

static make_hashable_helper(*obj*)
Recursively turns iterables into sorted tuples

static object_field_score(*collection*, *ioc*, *source_name*, *configuration_name*, *obj*, *objectId*, *expiry*, *max_expiry*, *field_set=None*)
Returns T2 average uniqueness

Takes a dictionary and returns the likelihood of that object being unique based on data in the collection

Parameters

- **collection** (*str*) – Deduplication collection name
- **ioc** (GreaseContainer) – IoC Access
- **source_name** (*str*) – source of data to be deduplicated
- **configuration_name** (*str*) – configuration name to be deduplicated
- **obj** (*dict*) – Single dimensional list to be compared against collection
- **objectId** (*str*) – T1 Hash Mongo ObjectId to be used to associate fields to a T1
- **expiry** (*int*) – Hours for deduplication to wait before removing a field if not seen again
- **max_expiry** (*int*) – Days for deduplication to wait before ensuring object is deleted
- **field_set** (*list, optional*) – List of fields to deduplicate with if provided. Else will use all keys

Returns Duplication Probability

Return type float

static string_match_percentage(*constant*, *new_value*)
Returns the percentage likelihood two strings are identical

Parameters

- **constant** (*str*) – Value to use as base standard
- **new_value** (*str*) – Value to compare *constant* against

Returns Percentage likelihood of duplicate value

Return type float

The Scheduling Engine

class tgt_grease.enterprise.Model.**Scheduling**(*ioc=None*)
Bases: object

Central scheduling class for GREASE

This class routes data to nodes within GREASE

ioc
IoC access for DeDuplication

Type *GreaseContainer*

determineDetectionServer()
Determines detection server to use

Finds the detection server available for a new detection job

Returns MongoDB Object ID of server & current job count

Return type tuple

determineExecutionServer(*role*)
Determines execution server to use

Finds the execution server available for a new execution job

Returns MongoDB Object ID of server; if one cannot be found then string will be empty

Return type str

determineSchedulingServer()
Determines scheduling server to use

Finds the scheduling server available for a new scheduling job

Returns MongoDB Object ID of server & current job count

Return type tuple

scheduleDetection(*source*, *configName*, *data*)
Schedule a Source Parse to detection

This method will take a list of single dimension dictionaries and schedule them for detection

Parameters

- **source** (*str*) – Name of the source
- **configName** (*str*) – Configuration Data was sourced from
- **data** (*list[dict]*) – Data to be scheduled for detection

Returns Scheduling success

Return type bool

scheduleScheduling(*objectId*)
Schedule a source for job scheduling

This method schedules a source for job scheduling

Parameters **objectId** (*str*) – MongoDB ObjectId to schedule

Returns If scheduling was successful

Return type bool

The Scanning Processor

class tgt_grease.enterprise.Model.**Scan**(*ioc=None*)
Bases: object

Scanning class for GREASE Scanner

This is the model to actually utilize the scanners to parse the configured environments

ioc
IOC for scanning

Type *GreaseContainer*

conf
Prototype configuration instance

Type *PrototypeConfig*

impTool
Import Utility Instance

Type *ImportTool*

dedup
Deduplication instance to be used

Type *Deduplication*

Parse(*source=None*, *config=None*)
This will read all configurations and attempt to scan the environment

This is the primary business logic for scanning in GREASE. This method will use configurations to parse the environment and attempt to schedule

---

Note: If a Source is specified then *only* that source is parsed. If a configuration is set then *only* that configuration is parsed. If both are provided then the configuration will *only* be parsed if it is of the source provided

---

Note: **If mocking is enabled**: Deduplication *will not occur*

Parameters

- **source** (*str*) – If set will only parse for the source listed
- **config** (*str*) – If set will only parse the specified config

Returns True unless error

Return type bool

static ParseSource(*ioc*, *source*, *configuration*, *deduplication*, *scheduler*)
Parses an individual source and attempts to schedule it

Parameters

- **ioc** (GreaseContainer) – IoC Instance
- **source** (BaseSourceClass) – Source to parse
- **configuration** (*dict*) – Prototype configuration to use
- **deduplication** (Deduplication) – Dedup engine instance
- **scheduler** (Scheduling) – Central Scheduling instance

Returns Meant to be run in a thread

Return type None

generate_config_set(*source=None*, *config=None*)
Examines configuration and returns list of configs to parse

Note: If a Source is specified then *only* that source is parsed. If a configuration is set then *only* that configuration is parsed. If both are provided then the configuration will *only* be parsed if it is of the source provided

Parameters

- **source** (*str*) – If set will only parse for the source listed
- **config** (*str*) – If set will only parse the specified config

Returns Returns Configurations to Parse for data

Return type list[dict]

The Detection Processor

class tgt_grease.enterprise.Model.**Detect**(*ioc=None*)
Bases: object

Detection class for GREASE detect

This is the model to actually utilize the detectors to parse the sources from scan

ioc
IOC for scanning

Type *GreaseContainer*

impTool
Import Utility Instance

Type *ImportTool*

conf
Prototype configuration tool

Type *PrototypeConfig*

scheduler
Prototype Scheduling Service Instance

Type *Scheduling*

detectSource()
This will perform detection the oldest source from SourceData

Returns If detection process was successful

Return type bool

detection(*source*, *configuration*)
Performs detection on a source with the provided configuration

Parameters

- **source** (*dict*) – Key->Value pairs from sourcing to detect upon
- **configuration** (*dict*) – Prototype configuration provided from sourcing

Returns Detection Results; first boolean for success, second dict of variables for context

Return type tuple

getScheduledSource()
Queries for oldest source that has been assigned for detection

Returns source awaiting detection

Return type dict

The Scheduling Processor

class tgt_grease.enterprise.Model.**Scheduler**(*ioc=None*)
Bases: object

Job Scheduler Model

This model will attempt to schedule a job for execution

ioc
IOC for scanning

Type *GreaseContainer*

impTool
Import Utility Instance

Type *ImportTool*

conf
Prototype configuration tool

Type *PrototypeConfig*

scheduler
Prototype Scheduling Service Instance

Type *Scheduling*

getDetectedSource()
Gets the oldest successfully detected source

Returns Object from MongoDB

Return type dict

schedule(*source*)
Schedules source for execution

Returns If scheduling was successful or not

Return type bool

scheduleExecution()
Schedules the oldest successfully detected source to execution

Returns True if detection is successful else false

Return type bool

Enterprise Prototypes

Sourcing Prototype (scan)

class tgt_grease.enterprise.Prototype.Scan.**Scanner**
Bases: tgt_grease.core.Types.Command.Command

The Scan Command

This class is the ingestion of information for GREASE. It utilizes configurations to 'wire' scanners together

execute(*context*)
Execute method of the scanner prototype

Parameters **context** (*dict*) – Command Context

Note: This method normally will *never* return. As it is a prototype. So it should continue into infinity

Returns True always unless failures occur

Return type bool

help = "\n This command scans the environment via the node configuration. This enables

purpose = 'Parse the configured environment for data and schedule de-duplicated data f

Detection Prototype (detect)

class tgt_grease.enterprise.Prototype.Detect.**Detection**(*Logger=None*)
Bases: tgt_grease.core.Types.Command.Command

The detect command

This class is the source detection for GREASE. It utilizes the logic block of your configurations to determine if a job needs to be run

execute(*context*)
Execute method of the detection prototype

Parameters **context** (*dict*) – Command Context

Note: This method normally will *never* return. As it is a prototype. So it should continue into infinity

Returns True always unless failures occur

Return type bool

help = '\n This command detects possible jobs from the scan command\n \n Args:\n --loo

purpose = 'Detect sources from scan and schedule them for job scheduling'

Scheduling Prototype (schedule)

class tgt_grease.enterprise.Prototype.Schedule.**Scheduling**(*Logger=None*)
Bases: tgt_grease.core.Types.Command.Command

The schedule command

This class is the job scheduling for GREASE. It utilizes the job and exe_env (if provided) keys of your configurations to schedule jobs for execution

execute (*context*)
Execute method of the scheduling

Parameters **context** (*dict*) – Command Context

Note: This method normally will *never* return. As it is a prototype. So it should continue into infinity

Returns True always unless failures occur

Return type bool

```
help = '\n This command schedules jobs for execution\n \n Args:\n --loop:<int>\n How ma
purpose = 'Schedule detected Jobs for Execution'
```

GREASE Sources

URL Parsing Source

class tgt_grease.enterprise.Sources.UrlParser.**URLParser**
Bases: tgt_grease.enterprise.Model.BaseSource.BaseSourceClass

Monitor URL's as a source of information

This source is designed to provide source data on the URL's configured for a GREASE sourcing cluster. A generic configuration looks like this for a url_source:

```
{
    'name': 'example_source', # <-- A name
    'job': 'example_job', # <-- Any job you want to run
    'exe_env': 'general', # <-- Selected execution environment; Can be anything!
    'source': 'url_source', # <-- This source
    'url': ['google.com', 'http://bing.com', '8.8.8.8'], # <-- List of URL's to␣
↪parse
    'hour': 16, # <-- **OPTIONAL** 24hr time hour to poll URLs
    'minute': 30, # <-- **OPTIONAL** Minute to poll URLs
    'logic': {} # <-- Whatever logic your heart desires
}
```

Note: This configuration is an example

Note: If a URL in the *url* parameter is not prefixed with *http://* then the class will do so for you

Note: without *minute* parameter the engine will poll for the entire hour

Note: **Hour and minute parameters are in UTC time**

Note: To only poll once an hour only set the **minute** field

mock_data(*configuration*)
Data from this source is mocked utilizing the GREASE Filesystem

Mock data for this source can be place in *<GREASE_DIR>/etc/*.mock.url.json*. This source will pick up all these files and load them into the returning object. They will need to follow this schema:

```
{
    'url': String, # <-- URL that would have been loaded
    'status_code': Int, # <-- HTTP Status code
    'headers': String, # <-- HTTP headers as a string
    'body': String # <-- HTTP response body
}
```

Parameters **configuration** (*dict*) – Configuration Data for source

Note: Argument **configuration** is not honored here

Returns Mocked Data

Return type list[dict]

parse_source(*configuration*)
This will make a GET request to all URL's in the list provided by your configuration

Parameters **configuration** (*dict*) – Configuration of Source. See Class Documentation above for more info

Returns If True data will be scheduled for ingestion after deduplication. If False the engine will bail out

Return type bool

ElasticSearch Source

class tgt_grease.enterprise.Sources.ElasticSearch.**ElasticSource**
Bases: tgt_grease.enterprise.Model.BaseSource.BaseSourceClass

Source data from ElasticSearch

This Source is designed to query ElasticSearch for data. A generic configuration looks like this for a elastic_source:

```
{
    'name': 'example_source', # <-- A name
    'job': 'example_job', # <-- Any job you want to run
    'exe_env': 'general', # <-- Selected execution environment; Can be anything!
    'source': 'elastic_source', # <-- This source
    'server': 'http://localhost:9200', # <-- String for ES Connection to occur
    'index': 'my_fake_index', # <-- Index to query within ES
    'doc_type': 'myData' # <-- Document type to query for in ES
    'query': {}, # <-- Dict of ElasticSearch Query
```

(continues on next page)

(continued from previous page)

```
    'hour': 16, # <-- **OPTIONAL** 24hr time hour to poll SQL
    'minute': 30, # <-- **OPTIONAL** Minute to poll SQL
    'logic': {} # <-- Whatever logic your heart desires
}
```

Note: without *minute* parameter the engine will poll for the entire hour

Note: **Hour and minute parameters are in UTC time**

Note: To only poll once an hour only set the **minute** field

mock_data(*configuration*)

Data from this source is mocked utilizing the GREASE Filesystem

Mock data for this source can be place in *<GREASE_DIR>/etc/*.mock.es.json*. This source will pick up all these files and load them into the returning object. The data in these files should reflect what you expect to return from ElasticSearch

Parameters **configuration** (*dict*) – Configuration Data for source

Note: Argument **configuration** is not honored here

Returns Mocked Data

Return type list[dict]

parse_source(*configuration*)

This will make a ElasticSearch connection & query to the configured server

Parameters **configuration** (*dict*) – Configuration of Source. See Class Documentation above for more info

Returns If True data will be scheduled for ingestion after deduplication. If False the engine will bail out

Return type bool

SQL Source

class tgt_grease.enterprise.Sources.SQLSearch.**SQLSource**

Bases: tgt_grease.enterprise.Model.BaseSource.BaseSourceClass

Source data from a SQL Database

This Source is designed to query a SQL Server for data. A generic configuration looks like this for a sql_source:

```
{
    'name': 'example_source', # <-- A name
    'job': 'example_job', # <-- Any job you want to run
    'exe_env': 'general', # <-- Selected execution environment; Can be anything!
```

(continues on next page)

(continued from previous page)

```
    'source': 'sql_source', # <-- This source
    'type': 'postgresql', # <-- SQL Server Type (Only supports PostgreSQL␣
↪Currently)
    'dsn': 'SQL_SERVER_CONNECTION', # <-- String representing the Environment␣
↪variable used to connect with
    'query': 'select count(*) as order_total from orders where oDate::DATE =␣
↪current_data', # <-- SQL Query to execute on server
    'hour': 16, # <-- **OPTIONAL** 24hr time hour to poll SQL
    'minute': 30, # <-- **OPTIONAL** Minute to poll SQL
    'logic': {} # <-- Whatever logic your heart desires
}
```

Note: This configuration is an example

Note: Currently We only support PostreSQL Server

Note: without *minute* parameter the engine will poll for the entire hour

Note: **Hour and minute parameters are in UTC time**

Note: To only poll once an hour only set the **minute** field

mock_data (*configuration*)

Data from this source is mocked utilizing the GREASE Filesystem

Mock data for this source can be place in *<GREASE_DIR>/etc/*.mock.sql.json*. This source will pick up all these files and load them into the returning object. The data in these files should reflect what you expect to return from SQL:

```
{
    'column expected': 'value expected'
    ...
}
```

Parameters **configuration** (*dict*) – Configuration Data for source

Note: Argument **configuration** is not honored here

Note: A mock file should represent a single row

Returns Mocked Data

Return type list[dict]

parse_source(*configuration*)
This will Query the SQL Server to find data

Parameters **configuration** (*dict*) – Configuration of Source. See Class Documentation above for more info

Returns If True data will be scheduled for ingestion after deduplication. If False the engine will bail out

Return type bool

GREASE Source Detectors

Detectors are used to parse data from sourcing and determine if a job needs to be executed.

NOTE: If developing a new detector it is the accepted practice for all non-error log messages to be set to only print in *verbose* mode

Regex Detector

class tgt_grease.enterprise.Detectors.**Regex**(*ioc=None*)
Bases: tgt_grease.enterprise.Model.BaseDetector.Detector

Regular Expression Detector for GREASE Detection

A Typical Regex configuration looks like this:

```
{
    ...
    'logic': {
        'Regex': [
            {
                'field': String, # <-- Field to search for
                'pattern': String, # <-- Regex to perform on field
                'variable': Boolean, # <-- OPTIONAL, if true then create a␣
↪context variable of result
                'variable_name: String # <-- REQUIRED IF variable, name of␣
↪context variable
            }
            ...
        ]
        ...
    }
}
```

processObject(*source*, *ruleConfig*)
Processes an object and returns valid rule data

Data returned in the second parameter from this method should be in this form:

```
{
    '<field>': Object # <-- if specified as a variable then return the key->
↪Value pairs
    ...
}
```

Parameters

- **source** (*dict*) – Source Data
- **ruleConfig** (*list[dict]*) – Rule Configuration Data

Returns first element boolean for success; second dict for any fields returned as variables

Return type tuple

Exists Detector

class tgt_grease.enterprise.Detectors.**Exists**(*ioc=None*)

Bases: tgt_grease.enterprise.Model.BaseDetector.Detector

Field Existence Detector for GREASE Detection

A Typical Regex configuration looks like this:

```
{
    ...
    'logic': {
        'Exists': [
            {
                'field': String, # <-- Field to search for
                'variable': Boolean, # <-- OPTIONAL, if true then create a␣
↪context variable of result
                'variable_name: String # <-- REQUIRED IF variable, name of␣
↪context variable
            }
            ...
        ]
        ...
    }
}
```

processObject(*source*, *ruleConfig*)

Processes an object and returns valid rule data

Data returned in the second parameter from this method should be in this form:

```
{
    '<field>': Object # <-- if specified as a variable then return the key->
↪Value pairs
    ...
}
```

Parameters

- **source** (*dict*) – Source Data
- **ruleConfig** (*list[dict]*) – Rule Configuration Data

Returns first element boolean for success; second dict for any fields returned as variables

Return type tuple

Range Detector

class tgt_grease.enterprise.Detectors.**Range**(*ioc=None*)

Bases: tgt_grease.enterprise.Model.BaseDetector.Detector

Field Number Range Detector for GREASE Detection

A Typical Range configuration looks like this:

```
{
    ...
    'logic': {
        'Range': [
            {
                'field': String, # <-- Field to search for
                'min': Int/float, # <-- OPTIONAL IF max is set
                'max': Int/Float # <-- OPTIONAL IF min is set
            }
            ...
        ]
        ...
    }
}
```

Note: The min field is only required if max is not present. This would ensure the field is above the number provided

Note: The max field is only required if min is not present. This would ensure the field is below the number provided

Note: If both are provided then the field would need to be inside the range provided

Note: To find an exact number set min one below the target and max one above the target

processObject(*source*, *ruleConfig*)

Processes an object and returns valid rule data

Data returned in the second parameter from this method should be in this form:

```
{
    '<field>': Object # <-- if specified as a variable then return the key->
↪Value pairs
    ...
}
```

Parameters

- **source** (*dict*) – Source Data
- **ruleConfig** (*list[dict]*) – Rule Configuration Data

Returns first element boolean for success; second dict for any fields returned as variables

Return type tuple

range_compare(*field*, *LogicalBlock*)

Compares number range to a field

Parameters

- **field** (*int/float/long*) – field to compare
- **LogicalBlock** (*dict*) – Logical Block

Returns if the range is successful then true else false

Return type bool

DateRange Detector

class tgt_grease.enterprise.Detectors.**DateRange**(*ioc=None*)

Bases: tgt_grease.enterprise.Model.BaseDetector.Detector

Date Range Detector for GREASE Detection

A Typical DateRange configuration looks like this:

```
{
    ...
    'logic': {
        'DateRange': [
            {
                'field': String, # <-- Field to search for
                'min': DateTime String, # <-- OPTIONAL IF max is set
                'max': DateTime String, # <-- OPTIONAL IF min is set
                'format': '%Y-%m-%d', # <-- Mandatory via strptime behavior
                'variable': Boolean, # <-- OPTIONAL, if true then create a␣
↪context variable of result
                'variable_name: String # <-- REQUIRED IF variable, name of␣
↪context variable
            }
            ...
        ]
        ...
    }
}
```

Note: The min field is only required if max is not present. This would ensure the field is after the date provided

Note: The max field is only required if min is not present. This would ensure the field is before the date provided

Note: Change the format to any supported https://docs.python.org/2/library/datetime.html#strftime-and-strptime-behavior

Note: To find an exact date/time set min one below the target and max one above the target

processObject(*source*, *ruleConfig*)

Processes an object and returns valid rule data

Data returned in the second parameter from this method should be in this form:

```
{
    '<field>': Object # <-- if specified as a variable then return the key->
↪Value pairs
    ...
}
```

Parameters

- **source** (*dict*) – Source Data
- **ruleConfig** (*list[dict]*) – Rule Configuration Data

Returns first element boolean for success; second dict for any fields returned as variables

Return type tuple

timeCompare(*field*, *LogicalBlock*)

Compares a date to a range

Parameters

- **field** (*str*) – field to compare
- **LogicalBlock** (*dict*) – Logical Block

Returns if the range is successful then true else false

Return type bool

DateDelta Detector

class tgt_grease.enterprise.Detectors.**DateDelta**(*ioc=None*)

Bases: tgt_grease.enterprise.Model.BaseDetector.Detector

Date Delta Detector for GREASE Detection

This detector differs from DateRange as it is relative. In DateRange you can determine constant days whereas DateDelta can tell you how many days in the future or past a field is from the date either specified or the current date.

A Typical DateDelta configuration looks like this:

```
{
    ...
    'logic': {
        'DateDelta': [
            {
                'field': String, # <-- Field to search for
                'delta': String, # <-- timedelta key for delta range; Accepted␣
↪Values: weeks, days, hours, minutes, seconds, milliseconds, microseconds,
                'delta_value': Int, # <-- numeric value for delta to be EX: 1␣
↪weeks
                'format': '%Y-%m-%d', # <-- Mandatory via strptime behavior
                'operator': String, # <-- Accepted Values: < <= > >= = !=
                'direction': String, # <-- Accepted Values: future past
                'date': String, # <-- OPTIONAL, if set then operation will be␣
↪performed on this date compared to field
                'variable': Boolean, # <-- OPTIONAL, if true then create a␣
↪context variable of result
```

(continues on next page)

(continued from previous page)

```
                'variable_name: String # <-- REQUIRED IF variable, name of␣
↪context variable
            }
            ...
        ]
        ...
    }
}
```

Note: Change the format to any supported https://docs.python.org/2/library/datetime.html#strftime-and-strptime-behavior

Note: If date field not provided will be assumed to be UTC Time

processObject (*source*, *ruleConfig*)
Processes an object and returns valid rule data

Data returned in the second parameter from this method should be in this form:

```
{
    '<field>': Object # <-- if specified as a variable then return the key->
↪Value pairs
    ...
}
```

Parameters

- **source** (*dict*) – Source Data
- **ruleConfig** (*list[dict]*) – Rule Configuration Data

Returns first element boolean for success; second dict for any fields returned as variables

Return type tuple

timeCompare (*field*, *LogicalBlock*)
Compares a date to find a delta

Parameters

- **field** (*str*) – field to compare
- **LogicalBlock** (*dict*) – Logical Block

Returns if the range is successful then true else false

Return type bool

2.4 GREASE Cluster Management

2.4.1 Subpackages

GREASE Cluster Management Commands

Monitoring Prototype (monitor)

This command prototype is installed on all servers by default. It ensures the cluster remains healthy and that nodes that are no longer responding are culled from the environment

class tgt_grease.management.Commands.monitor.**ClusterMonitor**
Bases: tgt_grease.core.Types.Command.Command

Cluster Monitor to ensure nodes are alive

execute(*context*)
This method monitors the environment

An [in]finite loop monitoring the cluster nodes for unhealthy ones

Parameters **context** (*dict*) – context for the command to use

Returns Command Success

Return type bool

help = '\n Ensures health of GREASE cluster by providing a Prototype to\n scan the act

purpose = 'Control cluster health'

Administrative CLI (bridge)

This command gives users a place to interact with their cluster on the CLI

class tgt_grease.management.Commands.bridge.**Bridge**
Bases: tgt_grease.core.Types.Command.Command

CLI tool for cluster administration

The command palate is listed here:

```
Args:
    register
        register this node with a GREASE Cluster as provided in the configuration␣
↪file
    info
        --node:<ObjectID>
            !Optional! parameter to observe a remote node. Defaults to look at␣
↪self
        --jobs
            !Optional! if set will list jobs executed
        --pJobs
            !Optional! include Prototype Jobs in list of jobs
    assign
        --prototype:<string>
            !mandatory if assigning a prototype! prototype to assign
            !NOTE! THIS MUST BE SEPARATED BY COLON OR EQUAL SIGN
        --role:<string>
            !mandatory if assigning a role! role to assign
            !NOTE! THIS MUST BE SEPARATED BY COLON OR EQUAL SIGN
        --node:<ObjectID>
            !Optional! remote node to assign job to
    unassign
        --prototype:<string>
            !mandatory if unassigning a prototype! prototype to unassign
```

(continues on next page)

(continued from previous page)

```
            !NOTE! THIS MUST BE SEPARATED BY COLON OR EQUAL SIGN
        --role:<string>
            !mandatory if unassigning a role! role to unassign
            !NOTE! THIS MUST BE SEPARATED BY COLON OR EQUAL SIGN
        --node:<ObjectID>
            !Optional! remote node to unassign job to
    cull
        --node:<ObjectID>
            !Optional! parameter to cull a remote node. Defaults to look at self
    activate
        --node:<ObjectID>
            !Optional! parameter to activate a remote node. Defaults to look at␣
↪self
    --foreground
        If set will print log messages to the commandline
```

Note: This tool is ever evolving! If you need something more feel free to create an issue!

bridge
Model Instance

Type *BridgeCommand*

execute(*context*)
This method monitors the environment

An [in]finite loop monitoring the cluster nodes for unhealthy ones

Parameters **context** (*dict*) – context for the command to use

Returns Command Success

Return type bool

help = '\n CLI for administrators to manage GREASE Clusters\n \n Args:\n register\n re

purpose = 'Control node/cluster operations'

Management Models

The Cluster Health Management Model

class tgt_grease.management.Model.**NodeMonitoring**(*ioc=<tgt_grease.core.InversionOfControl.GreaseContainer object>*)

Bases: object

Monitors cluster nodes for unhealthy state

ioc
IoC Access

Type *GreaseContainer*

centralScheduler
Central Scheduling Instance

Type *Scheduling*

scheduler
Scheduling Model Instance

Type *Scheduler*

deactivateServer(*serverId*)
deactivates server from pool

Parameters **serverId** (*str*) – ObjectId to deactivate

Returns If deactivation is successful

Return type bool

getServers()
Returns the servers to be monitored this cycle

Returns List of servers

Return type list[dict]

monitor()
Monitoring process

Returns If successful monitoring run occurred

Return type bool

rescheduleDetectJobs(*serverId*)
Reschedules any detection jobs

Parameters **serverId** (*str*) – Server ObjectId

Returns rescheduling success

Return type bool

rescheduleJobs(*serverId*)
Reschedules any detection jobs

Parameters **serverId** (*str*) – Server ObjectId

Returns rescheduling success

Return type bool

rescheduleScheduleJobs(*serverId*)
Reschedules any detection jobs

Parameters **serverId** (*str*) – Server ObjectId

Returns rescheduling success

Return type bool

scanComplete()
Enters a completed source so that this local server is alive next run

This method is so that the server's 'heart' beats after each run. It will insert a completed SourceData document and increments the job counter in the JobServer Document

Returns Writes a MongoDB Document

Return type None

schedule_detection_orphans()

schedule_execution_orphans()

schedule_orphans()

schedule_scheduling_orphans()

serverAlive(*serverId*)
Checks to see if server is alive

This method checks if the serverID exists in the collection and determines if it's execution number has changed recently. If it is a newly configured node it will be added to the monitoring collection

Parameters **serverId** (*str*) – ObjectId of server

Returns If server is alive

Return type bool

The Bridge Management Model

class tgt_grease.management.Model.**BridgeCommand**(*ioc=None*)
Bases: object

Methods for Cluster Administration

imp
Import Tool Instance

Type *ImportTool*

monitor
Node Monitoring Model Instance

Type *NodeMonitoring*

action_activate(*node=None*)
activates server in cluster

Parameters **node** (*str*) – MongoDB ObjectId to activate; defaults to local node

Returns If activation is successful

Return type bool

action_assign(*prototype=None*, *role=None*, *node=None*)
Assign prototypes/roles to a node either local or remote

Parameters

- **prototype** (*str*) – Prototype Job to assign
- **role** (*str*) – Role to assign
- **node** (*str*) – MongoDB ObjectId of node to assign to, if not provided will default to the local node

Returns If successful true else false

Return type bool

action_cull(*node=None*)
Culls a server from the active cluster

Parameters **node** (*str*) – MongoDB ObjectId to cull; defaults to local node

action_info(*node=None*, *jobs=None*, *prototypeJobs=None*)
Gets Node Information

Parameters

- **node** (*str*) – MongoDB Object ID to get information about
- **jobs** (*bool*) – If true then will retrieve jobs executed by this node
- **prototypeJobs** (*bool*) – If true then prototype jobs will be printed as well

---

Note: provide a node argument via the CLI –node=4390qwr2fvdew458239

Note: provide a jobs argument via teh CLI –jobs

Note: provide a prototype jobs argument via teh CLI –pJobs

Returns If Info was found

Return type bool

action_register()
Ensures Registration of server

Returns Registration status

Return type bool

action_unassign(*prototype=None*, *role=None*, *node=None*)
Unassign prototypes to a node either local or remote

Parameters

- **prototype** (*str*) – Prototype Job to unassign
- **role** (*str*) – Role to unassign
- **node** (*str*) – MongoDB ObjectId of node to unassign to, if not provided will default to the local node

Returns If successful true else false

Return type bool

valid_server(*node=None*)
Validates node is in the MongoDB instance connected to

Parameters **node** (*str*) – MongoDB Object ID to validate; defaults to local node

Returns first element is boolean if valid second is objectId as string

Return type tuple

CHAPTER 3

What is GREASE?

GREASE is the automation platform for L2+ Operations at Target. It is designed for all levels of operations staff to enable rapid development of automation tools from simple CLI based apps for daily ops work all the way to auto-recovery software.

3.1 What Does GREASE Stands For?

- Guest
- Reliability
- Engineering
- Automated
- Service
- Engine

3.2 What is Guest Reliability Engineering?

GRE is the newest generation of operations at Target. We utilize Google's SRE as inspiration for our new vision to bring joy to our guests. We help provide support to product groups across Target as well as help build more reliable & scalable systems.

CHAPTER 4

Indices and tables

- genindex
- search

Index

Symbols

A

B

C

D

E

F

G

H

I

S

T

U

V

W