YALP Documentation

Release 0.1

Timothy Messier

Jun 10, 2016

Contents

Distributed log parsing and collection.

YALP is a log parsing pipeline written in python. It utilized Celery for stable and scalable distributed processing, is easy to configure, and customize and extend.

CHAPTER 1

Install and Configure

Brief install guide:

```
$ sudo apt-get install rabbitmq-server
$ virtualenv /srv/yalp_env
$ source /srv/yalp_env/bin/activate
(yalp_env) $ pip install yalp
```

/srv/yalp.yml:

```
# Celery configuration
broker_url: amqp://guest:guest@localhost:5672//
inputs:
  - file:
      path: '/var/log/nginx/access.log'
parsers:
  - grok:
      pattern: '%{COMBINEDAPACHELOG}'
  - timestamp:
      field: timestamp
  - goip:
      field: clientip
      geoip_dat: /usr/share/GeoLiteCity.dat
  - user_agent:
      field: agent
  - url:
      field: request
outputs:
  - elasticsearch:
      uri: http://localhost:9200
```

```
(yalp_env) $ yalp-inputs -c /srv/yalp.yml
(yalp_env) $ yalp-parsers -c /srv/yalp.yml
(yalp_env) $ yalp-outputs -c /srv/yalp.yml
```

Full Installation Guide

CHAPTER 2

Reference

Full Plugin Reference

2.1 Install Guide

YALP is designed to be installed on multiple servers, with different components running on separate machines. It can just as easily be installed on a single machine. This guide will show how to setup all components on a single host, but will also describe how the components could easily be distributed.

2.1.1 Celery Broker

Since YALP uses Celery for communication between components, a broker must be installed. For this guide, the default broker rabbitmq will be used.

Brokers

For more info on brokers see Celery's broker docs.

To install rabbitmq under Ubuntu.

```
$ sudo apt-get install rabbitmq-server
```

2.1.2 Installing YALP

For now the easiest way to YALP is installed in a virtualenv.

```
$ virtualenv /srv/yalp_env
$ source /srv/yalp_env/bin/activate
```

Then install via pypi using `pip` or `easy_install`.

```
(yalp_env) $ pip install yalp
```

The three components `yalp-inputs`, `yalp-parsers` and `yalp-outputs`, should now be accessible.

2.1.3 Configuration

YALP uses a single `YAML` configuration file for all three components. Generally the config file should be consistent throughout the infrastructure, with the exception of the `yalp-inputs` configuration, which should be specific to the host where the input is being collected.

The first section of the config file deals with Celery configuration.

```
# Celery configuration
broker_url: amqp://guest:guest@localhost:5672//
parser_queue: parsers
output_queue: outputs
parser_worker_name: parser-workers
output_worker_name: output-workers
```

broker_url This is the connection uri for connecting to the broker.

parser_queue This is the name of the queue that the Parsers will watch for tasks. This can be set to any name so that it is easily identifiable, especially if the broker is being used for other services. The default name is `parsers`.

output_queue This is the name of the queue that the Outputs will watch for tasks. This can be set to any name so that it is easily identifiable, especially if the broker is being used for other services. The default name is `outputs`.

parser_worker_name This is the name on the Parser processes so that can easily be identifies via tools like `ps`.

output_worker_name This is the name on the Output processes so that can easily be identifies via tools like `ps`.

The next section of the config is for plugin configuration.

```
# Plugin configuration
input_packages:
  - yalp.inputs
parser_packages:
  - yalp.parsers
output_packages:
  - yalp.outputs
```

Each option contains a list of python packages that contain plugin modules for the specific component. This allows to specifying custom or third-part plugins. The defaults are in the example above.

Next is the `inputs` section.

```
# Input configuration
inputs:
  - file:
      path: '/var/log/nginx/access.log'
```

This section contains a list of inputs to monitor for events. This example is set to monitor `/var/log/messages`. The `type` option limits what parsers and outputers will process this input. Only parsers are outputs that have the same `type` will process the message. The general format is as follows.

Options for pluings

See the *Full Plugin Reference* for options to the plugins.

```
inputs:
  - <module>:
      <option>: <value>
      ...
      <option>: <value>
```

```
  - <module>:
      <option>: <value>
      ...
      <option>: <value>
```

The last two sections are similar to the `inputs` section but are for configuring the `parsers` and `outputs`.

```
parsers:
  - grok:
      pattern: '%{COMBINEDAPACHELOG}'

outputs:
  - elasticsearch:
      uri: http://localhost:9200
```

This configures the parsers to pass the message to the outpers without modifing it. The message will then to output to mongodb running on the same machine.

2.2 Logging Configuration

Logging configuration is done in the same config file as other *Configuration*.

2.2.1 Simple Config Options

By default, YALP will log warnings and errors to the console. The log level and format can be changed using the following options:

Log format

See Python's LogRecord attributes for details on log formats.

```
log_level: 'WARN'
log_format: '%(name)s: %(levelname)s [%(module)s:%(lineno)s] %(message)s'
```

2.2.2 Advanced Configuration

YALP supports advanced loggging configuration through the `logging` configuration option. For example to set YALP to log to Sentry:

```
logging:
  version: 1
  disable_existing_loggers: false
  handlers:
    sentry:
      level: DEBUG
      class: 'raven.handlers.logging.SentryHandler'
      dsn: 'https://public:secret@example.com/1'
  loggers:
    yalp.inputs:
      handlers:
        - sentry
      level: WARN
```

```
    propagate: false
  yalp.parsers
    handlers:
      - sentry
    level: WARN
    propagate: false
  yalp.outputs
    handlers:
      - sentry
    level: WARN
    propagate: false
```

Note: The loggers `yalp.inputs`, `yalp.parsers`, and `yalp.outputs` will catch all log messages for the corresponding plugins. To capture all of YALP's logs, use the `yalp` logger.

2.3 Plugin Reference

YALP uses plugins for the three components.

Inputers Collect events from input sources, such as log files.

Parsers Process a raw input event and transform, extract, or modify the input into a more organized output.

Outputers Record the event into a persistant storage, such as a file or mongo database.

2.3.1 Inputers

Inputers collect events from input sources. The events are send to the parsers for proccessing, or if there are no parsers configured, the events are directly sent to the outputers. All input events are sent as a dictionary with the fields `hostname` and `message` which contain the hostname where the inputer was collected and the raw input from the source. Events can also have an optional `type` field used to filter events. Custom inputers can also add additional optional fields.

Example input event:

```
{
  'hostname': 'localhost',
  'message': '127.0.0.1 - - [13/Mar/2014:13:46:00 -0400] "GET / HTTP/1.1" 200 6301 "-" "Mozilla/5.0
  'time_stamp': '2014-03-13T13:46:00',
  'type': 'nginx',
}
```

Full List of Inputers

yalp.inputs.file

The file inputer creates an event for each line in the file. It continues to follow the file much like the unix `tail -F` command. It will correctly follow the file event if the inode changes, like from tools like logroate. Additionally, the inputer saves its last position in the file so it does not reprocess lines if the service is stopped/restarted.

This inputer supports the following configuration items:

path The path of the file to collect events from.

type The type of event for parsers/outputers to filter on.

Example configuration.

```
inputs:
  - file:
      path: /var/log/messages
```

yalp.inputs.log_handler

The log handler inputer is for use within Python's logging facility. Therefore it uses a different configuration method than normal YALP plugins. Instead of using the YALP *YAML* configuration file, this inputer is configured in another project's python logging configuration.

For example, in another Python project:

```
LOGGING = {
    'version': 1,
    'disabled_existing_loggers': False,
    'handlers': {
        'yalp': {
            'level': 'INFO',
            'class': 'yalp.inputs.log_handler.YalpHandler',
            'type': 'my_package_logs',
            'pipeline': {
                'broker_url': 'amqp://guest@guest@localhost:5672//',
            },
        },
    },
    'loggers': {
        'my_package': {
            'handlers': ['yalp']
            'level': 'INFO',
        },
    }
}
```

The handler accepts the following optional fields:

type The type of the event for the parsers/outputers to filter on.

pipeline A dictionary of YALP configuration settings. The `pipeline` option accepts the following fields:

> ***parsers*** Boolean option. If true events will be sent to the parsers, otherwise they will be sent directly to the outputers. Default is `False`.
>
> Additionaly, `pipeline` option for the handler accepts all of the YALP Celery *Configuration* settings, such as the `broker_url`, with the same defaults.

Then in the Python project, send a log message as follows:

```
logger.info('a log message', extra={'additional': 'data will be included'})
```

This will create an event like:

```
{
    'time_stamp': '2015-01-01T01:00:00',
    'message': 'a log message',
    'additional': 'data will be included',
    'logger': 'my_pacakge.my_module',
```

```
    'funcName': 'func_with_log',
    'levelname': 'INFO',
    'levelno': 20,
    'hostname': 'my_host',
}
```

2.3.2 Parsers

Parsers process an input event transforming the raw message into more useful and organized fields. Therefore the parsed event dictionary may contain any number of fields. Parsers should preserve the `hostname` and optional `type` field of an input event.

Example parsed event:

```
{
  'hostname': 'localhost',
  'remote_addr': '127.0.0.1',
  'time_stamp': '2014-03-13T13:46:00',
  'request': '/',
  'status': '200',
  'bytes_send': '6301',
  'user_agent': 'Mozilla/5.0 (X11; Linux x86_64; rv:24.0) Gecko/20100101 Firefox/24.0',
  'type': 'nginx',
}
```

Full List of Parsers

yalp.parsers.grok

Use grok to parse the event. Any matched fields from the grok pattern will be added to the event.

This parser supports the following connfiguration items:

pattern A grok pattern to match. See available patterns for details.

field The field from the event to parse. Defaults to `message`.

type A type filter. Events not of this type will be skipped.

Example configuration.

```
parsers:
  - grok:
    pattern: '%{IP:ip_addr} %{WORD:request_type} %{URIPATHPARAM:path}'
```

With an input event like the following:

```
{
    'message': '192.168.0.1 GET /index.html',
    'time_stamp': '2015-01-01T01:00:00',
    'hostname': 'server_hostname',
}
```

After the parser runs, the event will become:

```
{
    'message': '192.168.0.1 GET /index.html',
    'time_stamp': '2015-01-01T01:00:00',
```

```
    'hostname': 'server_hostname',
    'ip_addr': '192.168.0.1',
    'request_type': 'GET',
    'path': '/index.html',
}
```

yalp.parsers.geoip

Extract Geo location data from an IP address.

The parser supports the following configuration items:

Warning: This parser requires the pygeoip package. The pygeoip package uses MaxMind's GeoIP dat files to get geo info from IP addresses. See http://dev.maxmind.com/geoip/legacy/geolite/ for more info.

Note: The geohash package is nessecary for converting latitude/longitude into geohashes. If not installed, the parser will store the raw latitude and longitude.

geoip_dat Path to the MaxMind GeoIP City dat file.

field The field containing the IP address to parse. If the field is not found in the event, the event will be skipped. Defaults to `clientip`.

out_field The field to set the Geo data to. Defaults to `geoip`.

use_hash Store location as a geohash. Default is `True`. If set to `False` location will be stored as ['lat', 'lon'] pair. Ignored if geohash is not installed.

type A type filter. Events not of this type will be skipped.

Example configuration.

```
parsers:
  - geoip:
      field: 'clientip'
      geoip_dat: '/usr/share/GeoLiteCity.dat'
```

yalp.parsers.keyvalue

Extract key, value paired data.

The parser supports the following configuration items:

field The field containing the key value pairs to parse.

out_field Set this to a field to store the parsed pairs under. If not set, the new fields are added at the top level of the event.

sep The seperator between key and value in a pair. Defaults to `:`

pair_sep The seperator between pairs of key/values. Defaults to a single space.

type A type filter. Events not of this type will be skipped.

yalp.parsers.regex

The regex parser applies a regex to the message of an event. Any named components of the regex become new keys in the event dict with the matched strings becoming the values.

Note: The original message is removed from the event.

This parser supports the following configuration items:

regex The regex to apply.

type A type filter. Only apply the regex to events of this type.

Example configuration.

```
parsers:
  - regex:
      regex: '(?P<month>\w+)\s+(?P<day>\d+)'
```

yalp.parsers.timestamp

Used to set the event time_stamp from another field in the event.

The parser supports the following configuration items:

field The field to parse for a datetime. If the field is not found in the event, the event will be skipped.

out_field The field to write the parsed time stamp to. Defaults to `time_stamp`.

timestamp_fmt The date format string to format the time stamp. Defaults to `%Y-%m-%dT$H:%M:%S`.

to_utc Convert the timestamp to UTC after parsing. Defaults to `True`.

type A type filter. Events not of this type will be skipped.

Examaple configuration.

```
parsers:
  - timestamp:
      field: date_field
```

yalp.parsers.transform

Used to convert a field in the event to a different built-in type.

The parser supports the following configuration items:

..note:: If the field fails to be transformed, The parser will log an error and leave the field as it was originally.

field The field to convert. If the field is not found in the event, the event will be skipped.

to Convert the field into this type. Supported types are:

```
    int
    float
    str
```

type A type filter. Events not of this type will be skipped.

Example configuration.

```
parsers:
  - transform:
      field: response_time
      to: int
```

yalp.parsers.user_agent

Extract browser, OS, device and other information from a user agent string.

The parser supports the following configuration items:

field The field containing the user agent string to parse. If the field is not found in the event, the event will be skipped. Defaults to `agent`.

out_field Set this to a field to store the user agent information under. If not set, the new field are added at the top level of the event.

type A type filter. Events not of this type will be skipped.

Example configuration.

```
parsers:
  - user_agent:
      field: 'agent'
      out_field: 'user_agent'
```

With an input event like the following:

```
{
    'hostname': 'server_hostname',
    'time_stamp': '2015-01-01T01:00:00',
    'message': '"Mozilla/5.0 (X11; Linux x86_64; rv:38.0) Gecko/20100101 Firefox/38.0"',
    'agent': '"Mozilla/5.0 (X11; Linux x86_64; rv:38.0) Gecko/20100101 Firefox/38.0"',
}
```

After the parser runs, the event will become:

```
{
    'hostname': 'server_hostname',
    'time_stamp': '2015-01-01T01:00:00',
    'message': '"Mozilla/5.0 (X11; Linux x86_64; rv:38.0) Gecko/20100101 Firefox/38.0"',
    'agent': '"Mozilla/5.0 (X11; Linux x86_64; rv:38.0) Gecko/20100101 Firefox/38.0"',
    'user_agent': {
        'os': {
            'family': 'Linux',
            'version': ''
        },
        'browser': {
            'family': 'Firefox',
            'version': '38'
        },
        'device': {
            'brand': None,
            'family': 'Other',
            'model': None,
        },
        'is_bot': False,
        'is_mobile': False,
        'is_pc': True,
```

```
        'is_tablet': False,
        'is_touch_capable': False,
    },
}
```

yalp.parsers.url

Extract components of a url.

The parser supports the following configuration items:

field The field containing the url string to parse. If the field is not found in the event, the event will be skipped. Defaults to `request`.

out_field The field to set the url components to. Defaults to `url`.

type A type filter. Events not of this type will be skipped.

Example configuration.

```
parsers:
  - url:
      field: 'request'
```

With an input event like the following:

```
{
    'hostname': 'server_hostname',
    'time_stamp': '2015-01-01T01:00:00',
    'request': '/index.html?param1=val1&param2=val2',
}
```

After the parser runs, the event will become:

```
{
    'hostname': 'server_hostname',
    'time_stamp': '2015-01-01T01:00:00',
    'request': '/index.html?param1=val1&param2=val2',
    'url': {
        'fragment': '',
        'hostname': None,
        'netloc': '',
        'params': '',
        'password': None,
        'path': '/index.html',
        'port': None,
        'query': {
            'param1': ['val1'],
            'param2': ['val2'],
        },
        'scheme': '',
        'username': None
    },
}
```

2.3.3 Outputers

Outputers record an event to persistant storage, such as a database or file.

Full List of Outpuers

yalp.outputs.file

The file outputer writes events to a file. Each event is recorded as a JSON string.

This outputer supports the following configuration items:

path The path of the file to write the events.

type A type filter. Only output events of this type.

Example configutation.

```
outputs:
  - file:
      path: /var/log/all_messages
```

yalp.outputs.mongo

The mongo outputer sends events to a mongo collection. Each event is recorded as a new document in the collection.

> **Warning:** This requires the pymongo pacakge to be installed.

This outputer supports the following configuration items:

uri The mongodb connection uri. Formatted as `mongodb://[user:password@]<host>[:port]/[auth_database]`

database The database name to store the documents.

collection The collection name.

type A type filter. Only output events of this type.

Example configuration.

```
outputs:
  - mongo:
      uri: 'mongodb://localhost:27017/yalp'
      database: yalp
      collection: logs
```

yalp.outputs.elasticsearch

The elasticsearch outputer sends events to an elasticsearch index.

> **Warning:** This requires the pyelasticsearch pacakge to be installed.

This outputer supports the following configuration items:

uri The elasticsearch connection uri Formatted as `http[s]://[user:password@]<host>[:port]/[path]`. Can also be a list of connection uris. Defaults to `http://localhost:9200/`.

index The index name to store the documents. Default to `yalp-%Y.%m.%d`. The index can contain a date format string for a dynamic index.

doc_type The document name. Default to `logs`.

time_based If the index is time based. This requires that the index name contains a date format string and that the event contains a valid time stamp. Default to `True`.

time_stamp_fmt The date format of the time stamp in the event. Not used if the `time_stamp` field is a datetime. Default to `%Y-%m-%dT%H:%M:%S`.

manage_template Allow yalp to manage the elasticsearch index template. Default to `True`.

template_name The name of the index template to create. Default to `yalp`.

template_overwrite Allow yalp to write over any existing template. Default to `False`.

buffer_size The outputer will buffer this many events before sending them all to elasticsearch via a bulk insert. Default is `500`.

type A type filter. Only output events of this type.

Example configuration.

```
outputs:
  - elasticsearch:
      uri: 'http://localhost:9200/'
      index: "yalp-%Y.%m.%d"
      doc_type: logs
```

2.4 Custom Plugins

YALP allows for building custom plugins. This allows YALP to be extended to support new input sources, output types, or custom parsing. Plugins are written in python and involve inheriting from a base class.

2.4.1 Custom Inputers

Inputers must inherit from `BaseInputer` and must implement the `run` function. The class name must be `Inputer` for YALP's plguin import system to discover the inputer. The module name will be used to configure the inputer. The `BaseInputer` provides a property `stopped` that should be used by `run` to stop collecting events and trigger a cleanup of resources. It also provides the function `enqueue_event(event)` that takes an event dictionary, adds the `hostname` and `type` fields and sends the event to the next phase.

Example Inputer `custominputer.py`:

```
from yalp.inputs import BaseInputer


class Inputer(BaseInputer):

    def __init__(self, custom_option, *args, **kwargs):
        super(Inputer, self).__init__(*args, **kwargs)
        self.custom_option = custom_option

    def _collect_event(self):
        # Custom event collection code. Returns a dictionary with key
        # `message` with value of raw input string.

    def run(self):
        # ... setup
        while not self.stopped:
            event = self._collect_event()
```

```
            self.enqueue_event(event)
        # ... cleanup
```

This inputer can then be configured `yalp.yml`:

```
input_packages:
  - yalp.inputs
  - package.with_custominputer_module

inputs:
  - custominputer:
      custom_option: 'option'
      type: 'custom'
```

2.4.2 Custom Parsers

Parsers must inherit from `BaseParser` and must implement the `parse` function. The class name must be `Parser` for YALP's plugin import system to discover the parser. The module name will be used to configure the parser. The `BaseParser` is written so that the `parse` function will only be called if the event passes the `type` filter, thus `parse` can assume it is ment to parse the event. The event will be a dict containing the `hostname`, `type`, and `message` keys. The `parse` function should remove the `message` from the event dict, parse the message and set new key/values into the event to sent to the outputer.

Example Parser `customparser.py`:

```
from yalp.parsers import BaseParser


class Parser(BaseParser):
    def __init__(self, custom_option, *args, **kwargs):
        super(Parser, self).__init__(*args, **kwargs)
        self.custom_option = custom_option

    def parse(self, event):
        message = event.pop('message')
        # ... parse message and set new fields into event
        return event
```

This parser can then be configured `yalp.yml`:

```
parser_packages:
  - yalp.parsers
  - package.with_customparser_module

parsers:
  - customparser:
      custom_option: 'option'
      type: 'custom'
```

2.4.3 Custom Outputers

Outputers must inherit from `BaseOutputer` and must implement the `output` and `shutdown` functions. The class name must be `Outputer` for YALP's plugin import system to discover the outputer. The module name will be used to configure the outputer. The `BaseOutpuer` is written so that the `output` function is only called if the event passes the `type` filter, thus `output` can assume the event should be output. The `shutdown` function is called when the service is stopped. It should preform and cleanup, cleanly releasing any resources.

Example outputer `customoutputer.py`:

```
from yalp.outputs import BaseOutputer


class Outputer(BaseOutputer):
    def __init__(self, custom_option, *args, **kwargs):
        super(Outputer, self)__init__(*args, **kwargs)
        self.resource = connect(custom_option)  # connecting to custom output service/database/source

    def output(self, event):
        self.resource.insert(event)  # send event to service/database/source

    def shutdown(self):
        self.resource.flush()  # ensure data is written
        self.resource.close()  # cleanup connection.
```

This outputer can then be configured `yalp.yml`:

```
output_packages:
  - yalp.outputs
  - package.with_customoutputer_module

outputs:
  - customoutputer:
      custom_option: 'option'
      type: 'custom'
```

2.4.4 Logging in Custom PLugins

All `Base*` plugin classes have logging already setup.This ensures that log messages are correctly routed based on the components. Log messages by using:

```
self.logger.warning('Warning message')
```

2.5 Scaling YALP

YALP is built to easily scale. It leverages the stability of Celery for distributed processing.

2.5.1 Parser Scaling

Parsers run as Celery workers. The workers run concurrent processes. The number of processes can be configured with the `parser_workers` option (default is 5). Additionally multiple `yalp-parsers` processes can be started on separate hosts. Ensure that each server uses the same YALP config file and has access to the broker.

2.5.2 Output Scaling

Outputers can scale in the same manner as parsers. Use the `output_workers` option (default is 1) and/or start multiple `yalp-putputs` processes on separate servers.

> **Warning:** Be sure that the configured outputers can handle concurrent output. Most databases like Mongo and Elasticsearch can, but the File outputer may garble the output.

y

Y