Parser Plugins using the opEvents object

Table of Contents

Creating an opEvents Object inside a parser plugin

opEvents Object is the object core to process and parse all the events.

This example shows how we can create the opEvents object for use inside a parser plugin.

If you have opEvents installed in another location dir=> "/usr/local/omk/conf" will need to change to reflect the full path of your conf directory

Minimal Example

```
package Event_State_Example;
our $VERSION="0.0.0";

use lib "/usr/local/omk/lib";
use strict;

use OMK::Common;
use OMK::opEvents;
use OMK::Log;
# arguments: the line (currently being parsed),
# and reference to the live event properties
# returns: (status-or-error)
#
# zero or undef: parsing for this event is aborted,
# and no event is created.
# 1: indicates success, event is created and changed event
# properties are incorporated.
# any other value: treated as error message, changed event
# properties are NOT incorporated but event parsing continues.


sub parse_enrich
{
        my ($line, $event) = @_;

        my $confCommon = loadOmkConfTable(conf=> "opCommon", dir=> "/usr/local/omk/conf");

        my $logger = OMK::Log->new(level => $confCommon->{"omkd_log_level"} || 'info',

path => $confCommon->{'<omk_logs>'}."/opEvents.log");

        my $OPE = OMK::opEvents->new(config => $confCommon,
                                     logprefix => "Plugin::Event_State_Example",
                                                  log => $logger);
        $OPE->getDb();

        $event->{Plugin_Used} = "Event_State_Example";

        return 1;
}

1;
```

opEvents-4.1.1 has a new interface to make working with the opEvents object easier, to retain backwards compat with your current plugins set set opevents_parser_plugin_use_events_obj: "true" in opCommon.json

opEvents-4.1.1 Events Object Interface

We have provided the current functions to make working with opEvents easier, we plan to add to this list overtime and if there is something your think is missing and could make working with opEvents easier please contact us.

function	description	version	args	returns
create_nmis_event	Creates a new event in nmis	4.1.1	node event element details level	event hash and error message or undef
create_event	Creates a new event in opEvents	4.1.1	event (hash)	event id and error message or undef
update_event	Updates a opEvents event	4.1.1	eventid (string) upddates (hash)	1 or 0 for success / failure and undef or error a error string
acknowledge_event	Acknowledges an event	4.1.1	eventid (string)	1 or 0 for success / failure and undef or error a error string

opEvents 3.2.4 Changes

From 3.2.4 opEvents sends an opEvents objects so all the resources can be reused in the plugin., in version 4.1.1 set opevents_parser_plugin_use_events_obj: "true" to retain functionality with $OPE in the below documentation

Following changes should be applied in the plugin:

Minimal Example

```
package Event_State_Example;
our $VERSION="0.0.0";

use lib "/usr/local/omk/lib";
use strict;

use OMK::Common;
use OMK::opEvents;
use OMK::Log;
# arguments: the line (currently being parsed),
# and reference to the live event properties
# returns: (status-or-error)
#
# zero or undef: parsing for this event is aborted,
# and no event is created.
# 1: indicates success, event is created and changed event
# properties are incorporated.
# any other value: treated as error message, changed event
# properties are NOT incorporated but event parsing continues.


sub parse_enrich
{
        my ($line, $event, $OPE) = @_;

        $event->{Plugin_Used} = "Event_State_Example";

        return 1;
}

1;
```

Much more clean and simple!

opEvents gt 3.2.4 - Sending an event to NMIS

It is also possible to send an event to NMIS using nmisx object:

```
    my $OPE = OMK::opEvents->new(config => $confCommon,
                                                                logprefix => "Plugin::TestPlugin, ",
                                                                log => $logger);
        $OPE->getDb();

        my $nmisx = $OPE->{nmisx};
        my $event = {
                node => "asgard-local",
                event => "Event from opEvents plugin",
                element => "plugin",
                details => "Event from opEvents plugin details",
                level => 8
        };
        my ($response, $eventobj) = $nmisx->createNmisEvent(event => $event);
        $OPE->log->info("MYParserPlugin:: Response: \n" . Dumper($response));
```

To close the event we should detect which is the close log and use the checkEvent method:

```
my $nmisnode = $OPE->{nmisx}->node($nodename);
Compat::NMIS::checkEvent(sys => $nmisnode->_Sys(), upevent => "Event Up",
                                                        node_uuid => $nmisnode->node_uuid, event => $event->
{name},
                                                        element => $event->{element} );
```

Getting events: getEventLogsModel

We can get a list of events using getEventLogsModel. This method takes the following arguments:

- log_name (=db collection, log_name can be: events, rawLogs, logArchive, or actionlog)
- time_start/end and/or a set of any of id, node_uuid or node_name type/element/details/action/archive/entry/acknowledged/escalate/priority /event_id to select events

Optional arguments:

- sort: mongo sort criteria
- limit: Return only N records at the most
- skip: skip N records at the beginning. Index N in the result set is at 0 in the response.
- paginate: sets the pagination mode, in which case the result array is fudged up sparsely to return 'complete' result elements without limit! - a dummy element is inserted at the 'complete' end, but only 0..limit are populated

getEventLogsModel will always return an array and should be expected to not have any values

In this example we are getting an event by its ID:

```
my $modelData = $OPE->getEventLogsModel(log_name => "events", id => '60516246c6c2b17094225a9c');
my $otherEvent = $modelData->[0];
#Lets set a new property on our newly parsed event to the nodes_uuid of a unrelated event.
$event->{other_event_nodeuuid} = $otherEvent->{node_uuid};
```

Sample data from the event with id 60516246c6c2b17094225a9c


```
{"_id":{"$oid":"60516246c6c2b17094225a9c"},"acknowledged":0,"action_checked":1,"actions":[{"action":"tag","
comment":"set to FALSE","date":"2021-03-18T10:49:59","details":"outageCurrent","event":"SNMP Down","node_uuid":"
3f49619e-b8ae-4e96-b56a-a7331baf71d3","time":1616028599}],"count":1,"date":"2021-03-18T10:48:28","
delayedaction":1616028598,"details":"get SNMP Service Data: No response from remote host \"13.56.2.146\"","
element":"","escalate":null,"event":"SNMP Down","friendly_acknowledged":0,"friendly_element":"","
friendly_escalate":"","host":"demo.opmantek.com","lastupdate":1616028599,"level":"Major","node":"demo.opmantek.
com","node_uuid":"3f49619e-b8ae-4e96-b56a-a7331baf71d3","nodeinfo":{"configuration_group":"DataCentre","
configuration_location":"test"},"priority":6,"state":"down","stateful":"SNMP","status_history":
[[1616028509.42444,null,"received",null],[1616028599.84117,null,"action_processing","complete"]],"
tag_outageCurrent":"FALSE","time":1616028508,"type":"nmis_eventlog"}
```


Getting multiple events

getEventLogsModel needs time_start and time_end if you are searching for events not by id, this is for safety and performance .

In this example we are looking for events with the name 'My_Monkey_Event' which have not been acknowledged and from the last 24 hours. These arguments are compounded into a AND query, some arguments are faster to find that others depending on indexes. If the query takes too long opEvents action parser might kill the script before anything is returned.

```
my $toBeAcknowledged = $OPE->getEventLogsModel(log_name => "events", event => 'My_Monkey_Event', acknowledged
=> 0,  time_start=> time - 86400, time_end => time);
foreach my $e (@{$toBeAcknowledged}){

}
```

Full list of arguments to search by

```
'_id' => $arg{id},
'time' => { '$gte' => $time_start, '$lt' => $time_end },
'event' => $arg{event},
'node_uuid' => $arg{node_uuid},
'type' => $arg{type},
'element' => $arg{element},
'details' => $arg{details},
'eventid' => $arg{event_id}, # only useful in actionlog
'action' => $arg{action}, # only useful in actionlog
'archive' => $arg{archive}, # only useful in archive log
'entry' => $arg{entry}, # only in raw log
'state' => $arg{state},
'nodeinfo.configuration.location' => {'$regex' => $arg{'nodeinfo.configuration_location'} || $arg{location}},
'nodeinfo.configuration.group' => {'$regex' => $arg{'nodeinfo.configuration_group'} || $arg{group}},
'acknowledged' => numify($arg{acknowledged}),
'escalate' => numify($arg{escalate}),
'priority' => numify($arg{priority}), });
```

Creating Events

opEvents object provides an easy way to create an event:

```
# Tell opEvents object to create the event
my ($error, $eventid) = $OPE->createEvent(event => $event);
```

It will return an error in case the event hasn't been created, or the eventId otherwise.

We first need to create the event. This is an example:

```
# Create Event Object
my $event = {};

my @maybeprops = grep($_ !~ /^(act|debug|quiet|conf|confdir|baseurl|username|password)$/, keys %$Q);
if (@maybeprops)
{
        my @rawinputs;
        for my $key (@maybeprops)
        {
                next if ($Q->{$key} eq ''); # supress empty properties
                die "argument $key cannt be set multiple times!\n" if (ref($Q->{$key}) eq "ARRAY");

                # ensure numeric values for anything numeric, or mongo comparison ops will
                # fail badly (especially true for time, which breaks the gui)
                $event->{$key} = numify($Q->{$key});
                push @rawinputs, "'$key=$Q->{$key}'";
        }
        $event->{entry} = join(" ", @rawinputs);
}
```

Updating Events

opEvents object also provides an easy way to update an event. We will need to pass the following arguments:

- _id: for identify the event to be updated
- _constraints: to disable db key munging

Everything else will be recorded as content, as-is, except "status_history" and "trigger_eventids":

- status_history: optional but special: must be array and this array will be ADDED to an existing status_history array.
- trigger_eventids: always saved as array, and a new value is ADDED.
- buttons: always saved as array, and a new value is ADDED.

returns undef if ok, error message otherwise (also logged)

As an example:

```
my $failure = $OPE->updateEvent( "_id" => $e->{_id},
                                                                priority => 5,
                                                                status_history => [ $now, $thisuser,
"priority", 5 ], );
```

Acknowledging an event

We can acknowledge an event by setting acknowledged => 1, and give it status history so we know who and when triggered the event to be acknowledged

```
my $now = time;
my $user = "parser_plugin";
my $failure = $OPE->updateEvent( "_id" => "60516246c6c2b17094225a9c",
  acknowledged => 1,
  status_history => [ $now, $thisuser, "acknowledged", 1 ], );
```

Putting all this together

```
package Event_State_Example;
our $VERSION="0.0.0";

use lib "/usr/local/omk/lib";
use strict;
#use func;
use OMK::Common;
use Data::Dumper;
use OMK::opEvents;
use OMK::Log;
# arguments: the line (currently being parsed),
# and reference to the live event properties
# returns: (status-or-error)
#
# zero or undef: parsing for this event is aborted,
# and no event is created.
# 1: indicates success, event is created and changed event
# properties are incorporated.
# any other value: treated as error message, changed event
# properties are NOT incorporated but event parsing continues.

sub parse_enrich
{
        my ($line, $event) = @_;

        my $confCommon = loadOmkConfTable(conf=> "opCommon", dir=> "/usr/local/omk/conf");

        my $logger = OMK::Log->new(level => $confCommon->{"omkd_log_level"} || 'info',

path => $confCommon->{'<omk_logs>'}."/opEvents.log");

        my $OPE = OMK::opEvents->new(config => $confCommon,
                                     logprefix => "Plugin::Event_State_Example",
                                                                log => $logger);
        $OPE->getDb();

        #We can get an event with an id
        my $modelData = $OPE->getEventLogsModel(log_name => "events", id => '60516246c6c2b17094225a9c');
        my $otherEvent = $modelData->[0];


        $event->{other_event_ack} = [];
        my $thisuser = "Plugin::Event_State_Example";
        #lets get an event by name and mark them acknowledged
        #you must pass time start and end if we are looking for events and not and event by an id

        #lets ack them
        foreach my $e (@{$toBeAcknowledged}){
                        my $now = time;
                        my $failure = $OPE->updateEvent( "_id" => $e->{_id},
                                                                                                                acknowledged =>
1,
                                                                                                        status_history => [
$now, $thisuser, "acknowledged", 1 ], );


                        push @{$event->{other_event_ack}}, $e->{_id}->to_string;
                        #TODO better error handling
                        return if($failure);
        }

        $event->{Plugin_Used} = "Event_State_Example";
        $event->{node} = "fulla-localhost";
        $event->{host} = "127.0.0.1";
        $event->{other_event} = $otherEvent->{_id}->to_string;
        return 1;
}

1;
```

Lookup node by node name

The first option, Build up a search hash and pass this to getEventLogsModel

```
my $search = {
    node => "MYNODENAME"
};
$OPE->getEventLogsModel(event => "SNMP Down", search => $search
```

The second option, there is a small internal helper on the opEvents object which will return the nodes UUID for a nodes name, it will be an array as it can return more than one node.

```
# small internal transition helper
# args: one node name
# returns: list of node uuids (as there can be more than one...)
my @node_uuids = $OPE->_node_to_uuids("MY_NODE_NAME");

$OPE->getEventLogsModel(event => "SNMP Down", node_uuid => @node_uuids[0]
```

Development Process for Events Plugins

opEvents is a real-time system so you need to develop the plugins with this in mind.

We cannot provide a direct command line tool to help, but we can help with a workflow for testing and developing Parser Plugins, this is also our workflow for EventActions.

For our workflow we have a script which appends an event to a file with a down event and the current time. Attached is a sample script which outputs a routing adjacency changed.

In the parse plugin you can use the opEvents logging object and setting the omkd_log_level to debug you can log what is happening.
The Data::Dumper object is great at dumping Perl structures so you can see exactly what is going on.

use Data::Dumper;
$OPE->log->debug("MYParserPlugin:: Dumping structure my_structure" . Dumper($my_structure));

After checking the logs you would then send an up event.

Process is:

- Edit plugin.
- Restart daemon
- Send “down” event.
- Check opEvents log
- Send “up” event.
- Check opEvents log
- Repeat as needed.