Event Actions and Escalation (opEvents 3)

opEvents provides the Event Action Policy as a flexible mechanism for reacting to events. This document briefly describes how to configure the service, the policy language and the currently supported actions.

Action Policy Language

The action policy is configured in `conf/EventActions.nmis`, primarily in the section named `policy`. The policy consists of any number of nested if-this-then-that clauses, which specify the conditions an event must conform to and what actions to take in case of a match. Further configuration sections specific to particular actions can be present in the same file.

Here is a brief example policy snippet:

```
'policy' => {
 '1' => {
         IF => 'node.configuration.customer eq "important"',
         THEN => { # a sub-policy
          '10' => {
                 IF => 'node.roleType eq "core" and event.event =~ qr{Down}',
                  THEN => ['log.disaster()', 'escalate.twentyfourseven()'], # the recommended format for
specifying actions is a list
                 BREAK => 'true'
           },
         '20' => {
                 IF => 'node.roleType eq "distribution" and event.event =~ qr{Down}',
                 THEN => 'priority(+2) AND email(admin)', # do avoid this legacy single-string format!
                 BREAK => 'false'
          },
         BREAK => 'false'
 },
 '2' => ...
},
```

The overall structure is relatively straight-forward: a rule has a numeric identifier which controls the order of evaluation, precisely one set of IF and THEN clauses and an optional BREAK property.

The IF expression is basically any arbitrary Perl expression, but tokens of the form `event.<name>`, `node.<name>` or `macro.<name>` are substituted with the respective event/node/macro property value. Macro properties are defined in the configuration file `opCommon.nmis` in the section `macro`. The special wildcards `event.any` and `node.any` are replaced by a logical true value. Furthermore, tokens that match `enrich.<extdb>.<queryname>.<column>` will be substituted with the result of an external enrichment query. In opEvents 3 all Perl operators, parentheses and so on can be used in IF expressions.

If your IF expression does require text that could be misinterpreted as a substitution token (e.g. the `"Nr.1"` in `IF => 'event.details eq "NTP Server Nr.1"'`), then you should escape the dotted expression with a backslash (e.g. `"NTP Server Nr\.1"`).

Please note that for maximum robustness you should express any regular expression in IFs as `/regexp contents/` or `qr{regexp contents}`, **NOT** as "regexp contents": the doublequoted variant only works for very simple patterns.

The substituted values are inserted into the expression in double quotes. In versions 2.0.4 and above, the special characters @%$" ` are backslash-escaped to ensure that Perl does not interpret them when the expression is evaluated. In version 2.0.4 and above, purely numeric values are inserted unquoted as they are; before they were inserted double-quoted like strings.

The standard event properties are listed on this page, and the common node properties are documented here.

The THEN clause is executed if and only if the IF expression evaluates as true (ie. non-zero, non-blank, defined). The THEN clause contains

- a nested sub-policy,
- or a list of one or more action invocations,
- or a single string that specifies any number of action invocations separated by the token " `AND` " (space, AND, space).
  This legacy format should be avoided. Providing a list of actions is both more robust and faster.

The order of action invocations is relevant: All given actions in a `THEN` clause will be executed in the given order and regardless of success or failure of prior ones.

All action invocations follow the same patterns: *`actionname(argument)`*, *`actionname.subtype()`* or *`actionname.subtype(argument)`*. Please note that the empty set of parentheses **must not** be omitted.

Policy evaluation starts at the outermost policy level, and proceeds in order of the numeric rule identifiers. All rules on the same nesting level are evaluated one after the other, unless a successful rule has its BREAK option set to true: in this case the rules after the successful one are skipped. No BREAK option present is interpreted as BREAK is false.

In the example above, rule 20 would be skipped if rule 10 succeeds, and policy evaluation would contine at rule 2. If rule 10's IF does not match, then its BREAK option has no effect. If the `IF` expression of rule 1 doesn't match, then the sub-policy 10/20 isn't considered at all.

Action Policy Application and Timing

Normally all newly created events are subject to policy actions immediately after having been created, but this can be fine-tuned and adjusted:

- No policy actions are performed for events with the property `action_checked` set to 1 or for events that are (already) acknowledged.
  The former can be controlled by custom parser rules, the latter is mostly affected by the configuration options `opevents_auto_acknowledge` and `opevents_auto_acknowledge_up`:
  With auto-acknowledge enabled, a stateful down event is automatically acknowledged when the corresponding up event arrives. In that case, the up event itself is also automatically acknowledged if and only if `opevents_auto_acknowledge_up` is set.
- If the configuration option `opevents_no_action_on_flap` is set to true in `conf/opCommon.nmis`, then no actions are performed on the down event that is involved in a flap event, and the down event is acknowledged. This is the default behaviour.
- Policy action handling is delayed by `state_flap_window` seconds for all stateful events, so that state flaps can be detected before any actions are performed.
- Policy action handling is delayed for synthetic events, if the event creation rule sets the property `delayedaction`.

Supported Policy Actions

Action Name	Description
log.*logtype*()	Log the event to a file, as plain text or in JSON format
script.*scriptname*()	Execute a user-defined script, possibly capturing the output
escalate.*policyname*()	Mark this event for escalation using a particular escalation policy
email(*contactname*)	Email the event details to a particular contact
syslog.*targetserver*(*prio*)	Send the event as Syslog message to a Syslog server, optionally overriding the event priority
nmissyslog.*targetserver*(*prio*)	Send the event as Syslog message to an NMIS Syslog server, in the format expected by NMIS
priority(*adjustment*)	Change the priority of the event Adjustment can be a number between 0 and 10 for fixed assignment, or +number or -number for relative adjustment.
tag.*tagname*(*value*)	Set a custom event property's value for static enrichment. Tagname is the name of the property to modify and must be a single string without spaces. Values are not restricted. (In the database the custom tag will be stored as "tag_*tagname*", hence you cannot overwrite opEvents-internal properties with this action. As a consequence, if your policy has IFs that need a tag's value, then these need to reference the tag with the 'long form' "tag_*tagname*".) In opEvents 2.0.2 and newer the tagname "`kb_topic`" is special and controls linking to external data sources.
acknowledge()	Acknowledges the event in question (which stops all escalation activity for the event). Supported in opEvents 2.0.3 and newer.

watchdog.set(*waittime*) watchdog.disable()	Creates or updates a watchdog timer for the node associated with the current event. The timer is set to expire in *waittime* seconds from now. If the timer is not disabled or updated before the expiration time, then a synthetic event named "Watchdog Timer expired" is generated. Note that all four watchdog actions are disabled if the current event itself is a watchdog expiration event.
element_watchdog.set(*waittime*) element_watchdog.disable()	Similar to the previous, but for watchdog timers that are specific to both the node and the element (e.g. an interface) of the current event. Element watchdog timers are independent of node watchdogs and of each other: Updating or disabling an element watchdog for say, `eth1` doesn't affect a timer for `lo0` for the same node.

Notes for watchdog and element_watchdog

The watchdog timer system does require a priming event to establish the timer in the first place, and if a timer is disabled using `watchdog.disable()` it is completely removed and forgotten.

The consequence of this design is that newly added nodes or elements are not subject to any watchdog timers until opEvent receives an event that causes the watchdog creation. This is normally not a problem, unless such a new node is not creating events because it is down for example. To create watchdog timers without or independent of an event, you can use opeventd's command line event creation facility.

Configuration for log.XYZ()

The `log` section of `conf/EventActions.nmis` controls the log types to be made available for actions. Here is an example:

```
'log' => {
 'tmp' => {
        format => 'text', # a simple textual log
         file => '/tmp/opevents.log', # last-or-all messages go into this file
         mode => 'append', # if not given, the log is overwritten,
    # the default template if none is set:
    # template => [ "event.time", "\t", "node.name", "\t", "event.event", "\t", "event.element",
    #               "event.state", "\t", "event.details" ],
 },
 'machinelog' => {
         format => 'json', # a machine-oriented json log
         dir => "/tmp/opevents_json", # the directory to log into (created on demand)
 },
},
```

You can setup any number of log types; just make sure that your log.*type*() action call uses the name of a defined log type.

Two log formats are supported, `text` and `json`.

- Text logs contain by default only the most essential event properties as a tab-delimited list, one event per line. If the `mode` argument is not present, then the log file is overwritten every time the action is executed; the more common mode would be `append`.
  In opEvents 3, `text` logs support output templates as shown in the example above; see the syslog.XYZ() section below for details about templates.
- JSON logs on the other hand contain all event properties, one event per JSON file. You have to give a `dir` option which specifies where those logfiles will be created. The logfiles are named *timestamp-number*.json, timestamp being the UNIX timestamp and number being a running counter (the UNIX timestamp has a one second granularity, number differentiates between multiple events in a single second).

Configuration for script.XYZ()

The script action lets you execute a program of your choice, and optionally captures and saves that program's output with the event. As usual, the section `script` of `conf/EventActions.nmis` contains the required configuration directives:

```
'script' => {
    'traceroute_node' => {
        arguments => '--max-hops=20 node.host',
        exec => '/bin/traceroute',
        output => 'save'
    },
    'ping_node' => {
        arguments => '-c 5 node.host',
        exec => '/bin/ping',
        output => 'save'
    },
    # supported since the 2016-11-01 rerelease of version 2.0.6
    'future_proof' => {
        exec => [ "/usr/local/bin/someprogram", "--first-fixed-arg", "no substitution happens here" ],
        arguments => [ "event.node", "event.event", "--extra", "event.details" ],
        output => "save",
        stderr => "save",
        exitcode => "save",
        max_tries => 2,
    },
 }
```

The path to the program file must be given in the `exec` option. Arguments can be passed to the program; simply add them to the `arguments` option. Any tokens of the form `event.`*name* or `node.`*name* will be replaced by the named event or node property, respectively. If the option `output` is set to `save`, then the output of the program execution is captured and saved with the event in question; otherwise the output is discarded.

Please Note:

- opEvents versions up to 2.0.3 do not support long-running programs in script actions, and opeventsd **blocks** until the action program terminates.
- From version 2.0.4 onwards, `script` action handling is asynchronous and parallel, and the event status gets updated whenever processing of a script action completes.
  Because of the asynchronous processing your action policy does not have access to any `script.<scriptname>` event properties.
- **Up to version 2.0.6**, script actions are excuted using the system shell.
  - As a consequence you have to ensure your script `arguments` are shell-safe, ie. that spaces are escaped or suitably quoted, that quotes line up and that the arguments do not contain unescaped shell metacharacters (",',`,!, &...).
  - The exit code of the external program is *not* captured, only its output on STDOUT (and STDERR, unless the `exec` argument disposes of STDERR explicitely with a "2>..." construct).
  - Argument substitution for `event.`*name* and `node.`*name* may need to be disabled (if your arguments need to contain a verbatim "event. sometext" string.
    This can be done by escaping the "." with an (escaped) backslash. For example

    ```
    arguments => 'node.host event\\.event ...and other stuff to feed the program'
    ```

    will cause the argument to contain the unsubstituted text 'event.event'. Node the use of single quotes.
- **Since the refresh of opEvents 2.0.6 on 2016-11-01**, script actions are no longer executed using a shell, but directly by `opeventsd` instead. This is much safer from a security perspective, and also generally faster.
  - It is **recommended** (but not required) that you change your script configuration to use the new list format for `arguments` (and `exec`), as shown in the example above (see "future_proof").
    If you use the list format, then each token is analysed for potential property substitution and then passed on to your program, separate from all other tokens.
    Spaces, backticks or other shell metacharacters are thus no longer problematic in an event or node property.
  - You can continue using the single-string `arguments` or `exec`, but then opEvents will perform the necessary word-splitting and *minimal* a mendments for backwards-compatibility only:
    If your `arguments` string contains quoted tokens like `"--some_program_arg=event.event"`, the surrounding double (or single) quotes are stripped.
    Please note that this is *not* performed for quotes anywhere else in your arguments string.
    I.e. with an arguments string like `--weird_argument=don't`, the single quote will be passed through to your program as-is.
  - If you need to disable substitution (to pass in strings like "event.sometext" verbiatim), escape the "." with a backslash.
    As a much better alternative you can also put verbatim arguments in the `exec` list, because only the `arguments` list is subject to substitution.
  - It is now possible to select whether the script exitcode should be captured and saved with the event.
    This is enabled by default, unless you add `exitcode` => '`false`' to your script configuration.
  - It is now also possible to select which combination of STDOUT and STDERR output of a script should be captured and saved.
    The config property `output` covers STDOUT, the property `stderr` STDERR. `stderr` defaults to the value of `output`, if not given explicitely.
    Adding `"2>&1"` to your script arguments is no longer supported.
  - Should you absolutely require shell features in your script action, simply use `/bin/sh` as the `exec` and set the `arguments` to your liking, but
    please note that this is substantially less secure than direct execution if `event.X` or `node.Y` substitutions are involved.
- opEvents version 2.2.2 and newer also supports the `max_tries` parameter which determines how often a failed script action may be retried; if `max_tries` is not set, then the default value 3 is used, i.e. up to three attempts to perform the action. Please note that action failure in this context means a script exceeding the maximum configured runtime or opEvents encountering a problem with starting the script, but *not* a script returning a nonzero exit code.

Configuration for email()

The action email is different from the others in that its configuration is stored in separate files: `conf/opCommon.nmis` sets the global email parameters, `conf/Contacts.nmis` contains the definitions of contacts that opEvents should know about, and `conf/EventEmails.nmis` defines which email template to use for a particular contact.

Here is an example `mail` section from `opCommon.nmis`:

```
'email' => {
 'mail_from' => 'yourmailname@yourdomain.com',
 # auth is attempted if both user and password are set
 'mail_user' => 'your_user_account@your_domain.com',
 'mail_password' => 'your_password',
 'mail_server' => 'smtp.yourdomain.com',
 'mail_server_port' => 25, # 487 is another common choice
 'mail_use_tls' => 'true', # use STARTTLS for encrypted smtp
},
```

At the very least you will have to set `mail_server` and `mail_server_port` to the appropriate values for your infrastructure; it is recommended that you use `mail_use_tls` so that emails (and username/password) are transmitted in encrypted form.

If your mail server requires smtp authentication for sending email, then set `mail_user` and `mail_password` to suitable values; It is also highly likely that you will need to adjust `mail_from` to a valid email address, which will be used as the sender's address.

The settings in `Contacts.nmis` are straight-forward and self-explanatory: a named contact section defines the name to use for the email action, and its `mail` attribute assigns one or more email addresses to this contact (multiple addresses must be given as a comma-separated string). At the current time opEvents uses only the `Email` part of `Contacts.nmis`.

Thus, to send event emails to contact `xyz` with email address `abc@def.com`, you have to specify the action as `email(xyz)`, and add a contact section for `xyz` (with email `abc@...`) to `Contacts.nmis`.

Please note that before version 2.0.4 `email` actions were handled synchronously and thus **blocked** the events processing until the email delivery concluded. In version 2.0.4 and newer this action is handled asynchronously in a separate process.

Configuration for syslog.XYZ() and nmissyslog.XYZ()

Actions that involve syslog servers require that `conf/EventActions.nmis` contains a matching server definition in its `syslog` section, similar to this example:

```
'syslog' => {
  'server1' => {
    'facility' => 'local1',
    'server' => 'localhost',    # default if not present
    'protocol' => 'udp',            # default if not present
    'port' => '514',                    # default if not present
    # default message format if no template given:
    # 'template' => [ "opEvents::", "macro.opevents_hostname", "event.time", ",", "node.name", ",",
    # "event.event", ",", "event.priority", ",", "event.element", ",", "event.details" ],
  },
  'server2' => {
    'facility' => 'local0',        # local0 is the default facility if none is given
     'server' => 'some.other.box',
     'protocol' => 'tcp',
     'format' => 'json',                # overrides template and causes the whole event to be logged as json
  },
},
# event forwarding via the syslog protocol, but in NMIS' slave_eventlog format
'nmissyslog' => {
  'slave' => {
    'facility' => 'daemon',
    'server' => "a.system.ready.for.this.slave.event.log",
    # default message format if no template given:
    # 'template' => [ "NMIS_Event::", "macro.opevents_hostname", "event.time", ",", "node.name", ",",
    #                                 "event.event", ",", "event.level", ",", "event.element", ",", "event.
details" ],
  },
},
```

The definition has to include the `server` name or address and the syslog `facility` to use; the `port` number defaults to 514, and at this time opEvents only supports syslog over udp `protocol`.
The syslog severity is computed from the event priority (see opEvents priority levels vs. NMIS and Syslog levels), or from the optional priority argument in the action call (e.g. `syslog.someserver(7)`).

In opEvents 3 the format and content of syslogged messages is customisable:

- If you set `format` to `json`, then the event is logged as JSON text. This format retains the event structure.
- If you set no `format`, then 'templated text' is used.
  You may provide a `template` list, which can consist of any number of strings; if a template string consists of `event.XYZ`, `node.ABC` or `macro.EFG` then the identified event, node or macro propertiy are substituted. Note that this is performed *only* if the template string contains nothing except the property indicator: e.g. "event.time" is subject to substitution, whereas "we saw event.details" is not.

The example above shows the default templates for the various syslog actions.

Configuration for escalate.XYZ()

Escalation of open issues is handled flexibly in opEvents: you can specify which events should be potentially escalated, and you can formulate different policies for those escalations. Escalations in opEvents apply only to unacknowledged events.

Writing `escalate.somepolicy()` in a `THEN` clause marks the matched event for future escalation according to the escalation rules of `somepolicy`. An event can be subject to multiple escalation policies at the same time. All escalation policies that an event is marked for will be applied independently, and when a policy is unapplicable because of time and day restrictions, it is ignored - but only temporarily until the time and day match up again.

Only when an event is acknowledged does escalation for it cease. Events are normally acknowledged manually, but for stateful entities the "down" event is acknowledged automatically if the configuration option `opevents_auto_acknowledge` is enabled in `conf/opCommon.nmis`.

Escalation Policies

To formulate an escalation policy, you need to decide on your preferred escalation steps, their respective time thresholds and actions, and express that in section `escalate` of the config file `conf/EventActions.nmis`. Here is an example configuration fragment:

```
'escalate' => {
 'weekday' => {
         'name' => 'weekday',
         'IF' => {
                 priority => '>= 0',
                 days => 'Monday,Tuesday,Wednesday,Thursday,Friday',
                 begin => '9:00',
                 end => '19:00',
         },
         '60' => 'log.problem() AND script.ping_node()',
         '300' => 'email(operations)',
         '1200' => 'email(operations_pager) AND script.disaster()',
         '2400' => 'email(operations_manager)',
         '3600' => 'email(it_manager)',
 },

 'afterhours' => {
...
```

Your escalation policy clearly needs a name; the example uses `weekday` and `afterhours`. The two other components of the escalation policy are the `IF` clause, which sets the scope of the policy, and the list of escalation steps.

Escalation Time and Priority Restrictions

The `IF` clause is used to determine whether a particular escalation policy should be active at a given time, and for events of a given priority.

The `priority` setting is required and contains a comparison operator, a space and a number.
If your policy is to be unrestricted simply use `>= 0` (event priorities range from 0 to 10).

The `days` setting is optional, and should contain a comma-separated list of weekdays when the policy should be active. If `days` are not given, then the policy works on all days.
The days must be given by their full names, ie. "Monday" or "Thursday".

The `begin` and `end` properties set up the daily time range for this policy. No `begin` means "starts at midnight" and no `end` is interpreted as "ends at midnight".
The policy will be active in the interval between begin and end, if the begin time is earlier than end (like in the example above).
To invert the interval meaning, ie. for events *outside* the given (business) hours, simply swap begin and end over. For example, a policy with begin 18:00 and end 05:00 will work after 18:00 and before 05:00.

All criteria must match for an escalation policy to be active.

Escalation Step Definition

The remaining components of the escalation policy are the definitions of the escalation steps; these consist of the escalation threshold, and the actions to take. The escalation threshold (in seconds) specifies the minimum age of the unacknowledged event for this escalation step to activate, and the action part works the same as the `THEN` expression in the action policy.

When escalations are processed, the highest new escalation step is determined based on the age of the event, the associated actions are performed and the event state is updated. When escalations are processed next, only escalation steps higher than the most recently active one will be considered for this event. Please note that different escalation polices are applied independently and each has its own active highest escalation step.

With the example `weekday` policy above, an event would be acted upon after 60 unacknowledged seconds, then again once it reaches 300 unacknowledged seconds and so on. Each action would be taken at most once: if the policy becomes active for the first time if the event is already 5900 seconds unacknowledged, then only the highest escalation step (3600) would be applied.

The action part of the step definition has the same syntax and interpretation as the `THEN` expressions of the main action policy described earlier in this document, except that action `escalate.anypolicy()` from within an escalation policy makes no sense and is therefore disabled.